

戴冠平 编著

WebLogic 企业级运维实战

清华大学出版社

叱咤风云：WebLogic 企业级运维实战

戴冠平　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

本书由浅入深地论述了 WebLogic 的体系和理念，结合作者多年从业经验，充分透彻的剖析了 WebLogic 的核心技术，对于大型 J2EE 中间件应用，给出了系统性的方案和建议。尤为重要的是，对于 WebLogic 进入实际生产应用这十多年中，客户系统累积出现的各种典型故障和错误，分门别类地进行了透彻讲解，给出了具体的诊断思路和解决方案，具有非常现实、非常重要的指导意义和实战价值。

全书共有 4 篇、33 章。本书适合作为 Weblogic 运维技术人员的参考手册，也可以作为高校相关专业师生的学习资料。

图书在版编目（CIP）数据

叱咤风云：WebLogic 企业级运维实战 / 戴冠平编著. —北京：清华大学出版社，2011.12
ISBN 978-7-302-26760-7

Ⅰ. ①叱…　Ⅱ. ①戴…　Ⅲ. ①互联网络－基本知识　Ⅳ. ①TP393.4

中国版本图书馆 CIP 数据核字（2011）第 185703 号

责任编辑：夏兆彦
责任校对：徐俊伟
责任印制：
出版发行：清华大学出版社　　　　**地　　址：**北京清华大学学研大厦 A 座
http://www.tup.com.cn　　　　**邮　　编：**100084
社 总 机：010-62770175　　　　**邮　　购：**010-62786544
投稿与读者服务：010-62795954，jsjjc@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：190×260　**印　　张：**25.75　　　**字　　数：**643 千字
版　　次：2011 年 12 月第 1 版　　　**印　　次：**2011 年 12 月第 1 次印刷
印　　数：
定　　价：　　元

产品编号：043635-01

前　　言

20 世纪 80 年代以来，伴随着网络和互联网的快速发展，中国各行各业的企业级信息系统应用从其体系结构、所采纳的技术和应用本身的复杂性都发生了深刻的变革。企业的 IT 系统日益成为其业务运营的支撑核心。主机、数据库、中间件、存储、网络等基础设施软硬件构成了企业 IT 系统的基石，同时这些不同层面的技术随着企业应用的规模扩张和复杂性增强，对于企业的 IT 运维团队提出了越来越高的挑战，这也促使企业用户越来越深刻地认识到运维和服务所起的关键作用，及其带给自己的不菲价值。

在当今信息时代和人才时代，如何能够获得高质量的、快捷便利的 IT 服务是摆在所有企业 IT 运维团队面前一个迫切需要思考的问题，而单纯求援于目前 IT 大厂商的昂贵服务，其成本常常令大家觉得突兀和吃惊。一个可以作为参考的业内公开数据是，主流企业级软件公司的主要收入，尤其是利润来源，并不是软件销售本身，而是其技术和运维服务；而一般只够支付几年的服务运维费用，就已足够购买一套全新的软件产品。如果有个机会能够以相对公开的方式，针对企业级 IT 的运维进行系统性的知识整理和经验共享，相信这种努力和转移应当是件相当有益处并有意义的工作。

从我个人来讲，毕竟做了这么多年的中间件，很久很久以前，就一直想写这么一本书和大家分享；毕竟曾经写了那么多零零散散的心得，如浩瀚夜空中的点点繁星，错落杂乱；终于有了一大段空闲的时光，能让这些点点滴滴串通起来，努力组成一个有形状有系统的星座。

感谢当年引导和伴随我成长的 BEA 老同事们，让我有机会在中间件的领域里遨游，越游越深，虽然现在应该改称 Oracle 的各位新朋旧友了。也特别感谢联动北方的技术团队，正是你们的敬业精神，才激励我最终坚持把这本书编著出来；也正是你们的辛勤帮助，才有了这本书如此专业翔实的内容。

独坐明窗映斜阳，听沧海潮起潮落，观碧空云卷云舒；恰逢母校百年风范依然，学子拳拳赤心依旧，“自强不息，厚德载物”。此书的及时完稿，也算是一丝慰藉和心意了。

作者

2011 年于 Brisbane Manly 海滨

目　　录

第 1 篇　入　门　篇

第 2 篇　基　础　篇

第 3 篇 实 施 篇

第 4 篇 诊 断 篇

第 1 篇

入 门 篇

第 1 章　WebLogic 概述

WebLogic 是美国 BEA 公司出品的一个中间件产品，是用于开发、集成、部署和管理大型分布式 Web 应用、网络应用和数据库应用的 JavaEE 应用服务器。WebLogic Server 拥有处理关键 Web 应用系统问题所需的性能、安全、可扩展性和高可用性，同时又易于安装、部署和管理。

1.1　什么是 WebLogic 系统

WebLogic 是目前主流 J2EE 服务器之一，支持符合 J2EE 标准的各类应用程序（Application），其主要类型如下。

- **Web 模块**　HTML 网页、Servlet、JSP 网页、有关的 Java 类、标准的 J2EE Web 配置文件、WebLogic 有关的配置文件，如 weblogic.xml 以及其他文件，如 XML 文件、图像文件等。
- **EJB 模块**　包含 session bean、entity bean、message-driven bean 等。
- **Connector Modules（连接器模块）**　用于和 EIS 交互的 Java 类，可能还有 Native Modules。
- **Enterprise Application（企业应用）**　作为一个整体，包含上述一个或者几个模块。

1.2　WebLogic 的历史及发展

1995 年，Paul Ambrose、Bob Pasker、Laurie Pitman 和 Carl Resnikoff 一起创立了 WebLogic 公司并研发出 WebLogic 应用服务器产品，BEA 公司于 1998 年完成了对 WebLogic 公司的收购，2008 年，国际软件巨头 Oracle（甲骨文）公司对 BEA 进行了收购。

Oracle 公司收购 BEA 公司以后，在 Oracle 融合中间件产品线与 BEA 产品的整合上，已经取得了里程碑式的关键进展，而 Oracle 公司不久前发布了 Oracle 融合中间件战略和产品路线图，Oracle 公司已经加快了对 SOA 的提供。通过 Oracle 应用网格架构独一无二的性能，Oracle 公司的客户可获得行业最完整、开放和集成的中间件架构。为使客户获得连续的产品和服务，Oracle 公司还成功整合了前 BEA 的员工，创建了一个统一和完整的开发团队，而这个团队在中间件领域具有无可比拟的技术优势。自 2008 年 7 月份以来，已经有 4000 多个客户和合作伙伴参加了 68 场旨在描述 Oracle 中间件产品蓝图的战略交流会和 BEA 客户欢迎活动。Oracle 融合中间件生态系统已经拥有超过 8 万个客户和 2 万家合作伙伴，包括系统集成商、独立软件开发商、增值分销商和增值经销商。由此可见，Oracle 公司对 BEA 公司的收购对以后的发展起到了关键性的作用。

目前 WebLogic 最新的版本是 Oracle WebLogic Server 11g Rel1。

1.3　WebLogic 支持的平台及数据库

WebLogic 支持分布式异构体系，能利用多种数据库平台并支持运行于多种操作系统，具体见表 1-1。

表 1-1

数据库	❑ Oracle（及 Oracle RAC） ❑ IBM DB2 ❑ Microsoft SQL Server ❑ MySQL ❑ Sybase
操作系统	❑ Linux ❑ AIX ❑ HP-UX ❑ Solaris ❑ Windows
Java	❑ Java 平台，标准版 6 ❑ Java 平台，企业版 5

1.4　WebLogic 10g/11g 新特性

Oracle WebLogic Server 10g R3 把开发人员和最终用户的体验提高到一个新的水平，这主要体现在提供了更轻量级的 footprint、可选服务启动、快速启动，另一方面，新的 Fast Swap 功能提供无缝和快速的开发—调试—测试周期。

Oracle WebLogic Server 10g R3 为 Spring 开发人员提供了进一步的支持，让基于 SSH 框架的应用可以充分利用 Oracle WebLogic 丰富的功能、高性能、稳定性和易管理性。

新的 HTTP 发布—订阅特性支持构建即时、动态更新的 Web 2.0 风格的富客户界面。

支持 Java SE 6 和 JRockit Mission Control，开发人员和最终用户可以充分利用这些条件施展才华，并且对应用程序的动态行为和性能做深入分析。

1.5　WebLogic 的技术架构

WebLogic Platform 由下列产品组成，如图 1-1 所示。

- ❑ **Oracle WebLogic Server**　WebLogic 应用服务器是整个 WebLogic 产品的核心。
- ❑ **Oracle WebLogic Workshop**　WebLogic 的集成开发环境。

- ❑ **Oracle WebLogic Portal** Weblogic 应用门户服务器。
- ❑ **OracleWebLogic Integration** Weblogic 应用集成服务器。
- ❑ **OracleWebLogic Jrockit** Weblogic 的 Java SDK，支持 Java SE 6。

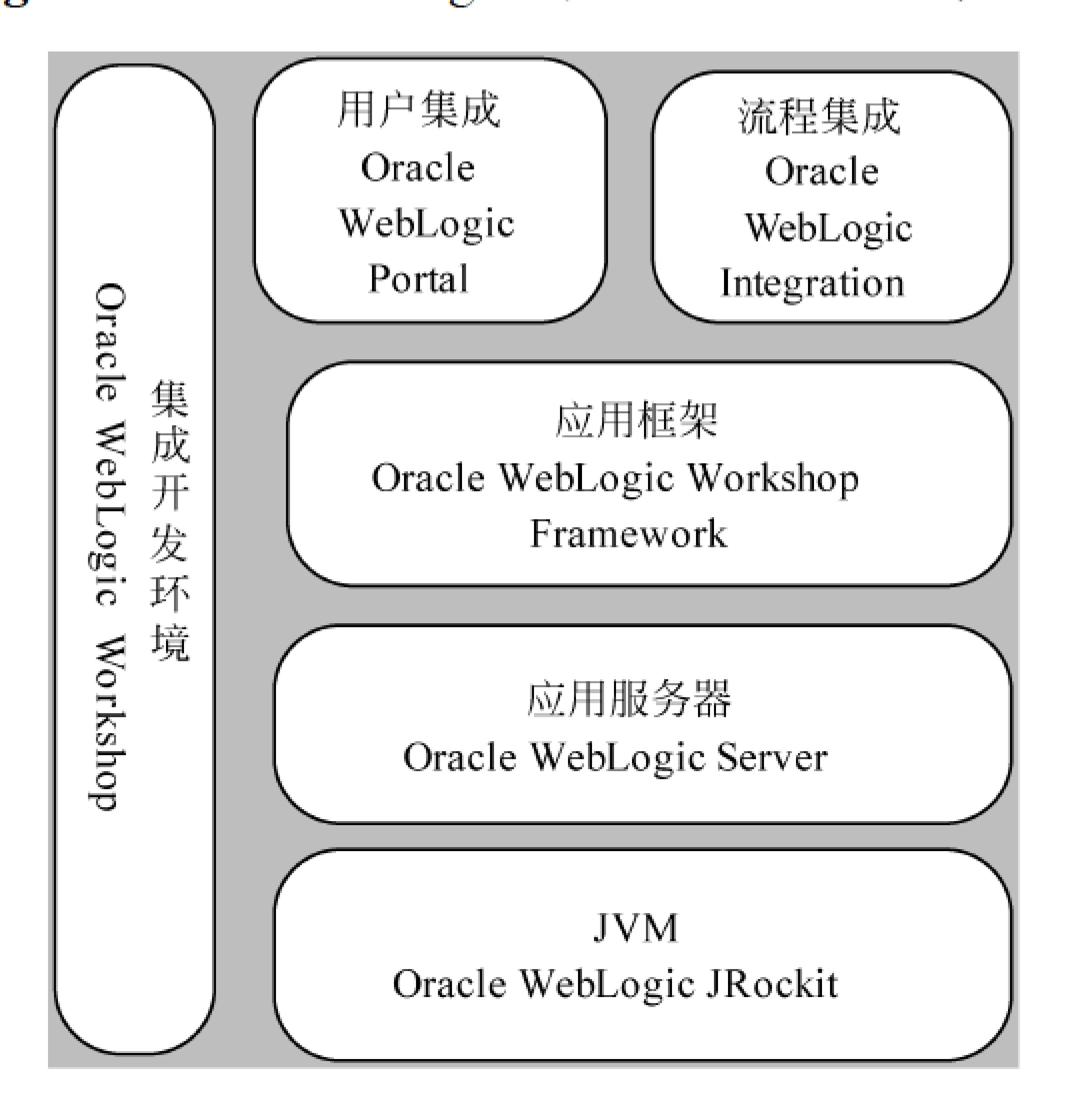


图 1-1

WebLogic 提供了一个文件，可安装所有这些产品；另外这些产品也有单独的安装文件，可以根据自己的需要分别安装。

1.6 WebLogic 系统的关键特性

WebLogic Server 拥有处理关键 Web 应用系统问题所需的多种特色和优势，具体体现在以下几方面。

（1）高扩展性。当系统的整体性能不能满足业务压力要求时，为了提高吞吐量，不需要做应用代码的修改，只要做系统横向或纵向的扩展，在集群中动态地添加新的 WebLogic Server 实例，部署相应的应用。这样可以充分利用现有设备，并保证了系统良好的扩展性。

（2）高可靠性。同样的服务可由集群中的多个 Server 来提供。

（3）高可用性。集群中不管是管理服务器还是被管服务器，在出现故障时都能保证应用的继续运行。

（4）高性能。对分布异构的支持，WebLogic 可以处理大量的并发访问。

1.7 WebLogic 与其他产品横向与纵向的比较

市场上应用比较广泛的类似产品还有 Tomcat、JBoss、IBM WebSphere。

Tomcat 是 Apache 项目开发的开源 Web 容器，只支持部分 J2EE 规范特性，例如

JSP/Servlet、JNDI 等，需要配合其他的组件实现特定的 J2EE 规范技术，例如通过集成 ActiveMQ 实现 JMS，通过 JOTM 实现 JTAG，等等。Tomcat 体积小、占用的资源小，处理能力也有限。一般初学者可以用它来调试 Web 应用，但用做商业企业级应用服务器就不太妥当了。

JBoss 不但是 Servlet 容器，也是 EJB 容器，是 J2EE 规范的完全实现，Web 容器部分通过集成 Tomcat 实现。

WebSphere 全面支持 J2EE 规范，依靠 IBM 在服务器上的市场优势也不断被广泛应用，性能和稳定性也较好，但易用性有待提高。

WebLogic 功能很强大，全面支持 J2EE 规范，有自己独到的核心技术，是一款十分强大的服务器软件，提供高可靠性、稳定性、可用性和高性能，安装、调试、配置优于前者，远程管理比较方便，是目前市场上占有率一直很高的服务器，在电力、电信、银行等大型企业中有着广泛的应用。

第 2 章　Windows 平台 WebLogic 的简单安装

2.1　安装前的准备工作

（1）安装 WebLogic，需要有与其版本相对应的 JDK（WebLogic 安装过程中会安装相应的 JDK，所以无需单独安装），以下是不同版本的对应。

- ❑ WebLogic 10 以上的版本要求 JDK1.6.0。
- ❑ WebLogic 9.2 以上的版本要求 JDK1.5.0。
- ❑ WebLogic 8.1 要求 JDK1.4.2。

（2）硬件要求：内存不少于 512MB，拟装目录下可用硬盘不少于 1GB。

2.2　安装 WebLogic 的详细步骤

以下提供了 WebLogic 10.3.3(11g)的安装步骤。

1．准备安装

双击安装程序，出现如图 2-1 所示的安装界面。

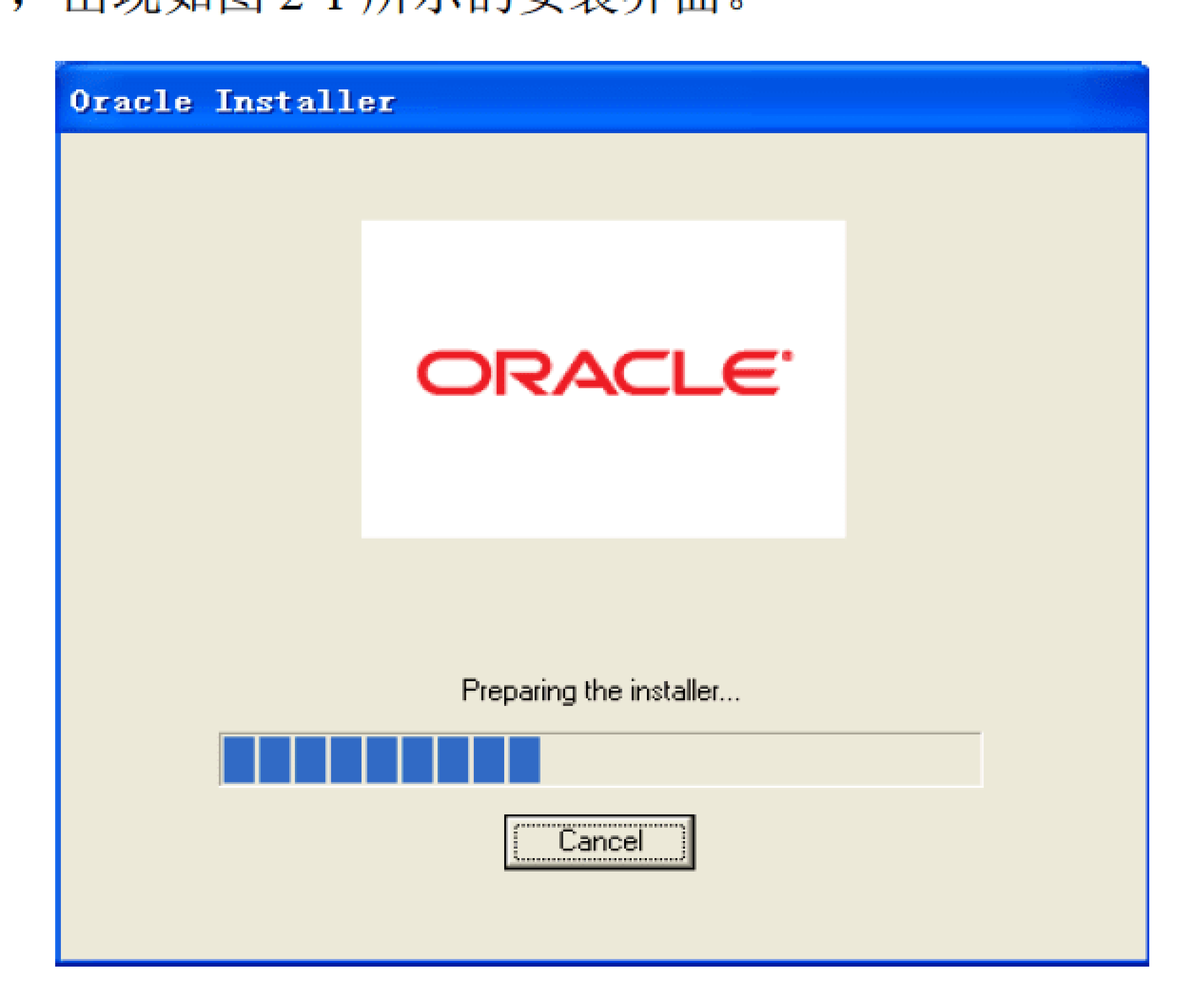


图 2-1

2．进入安装界面

进入安装界面如图 2-2 所示。

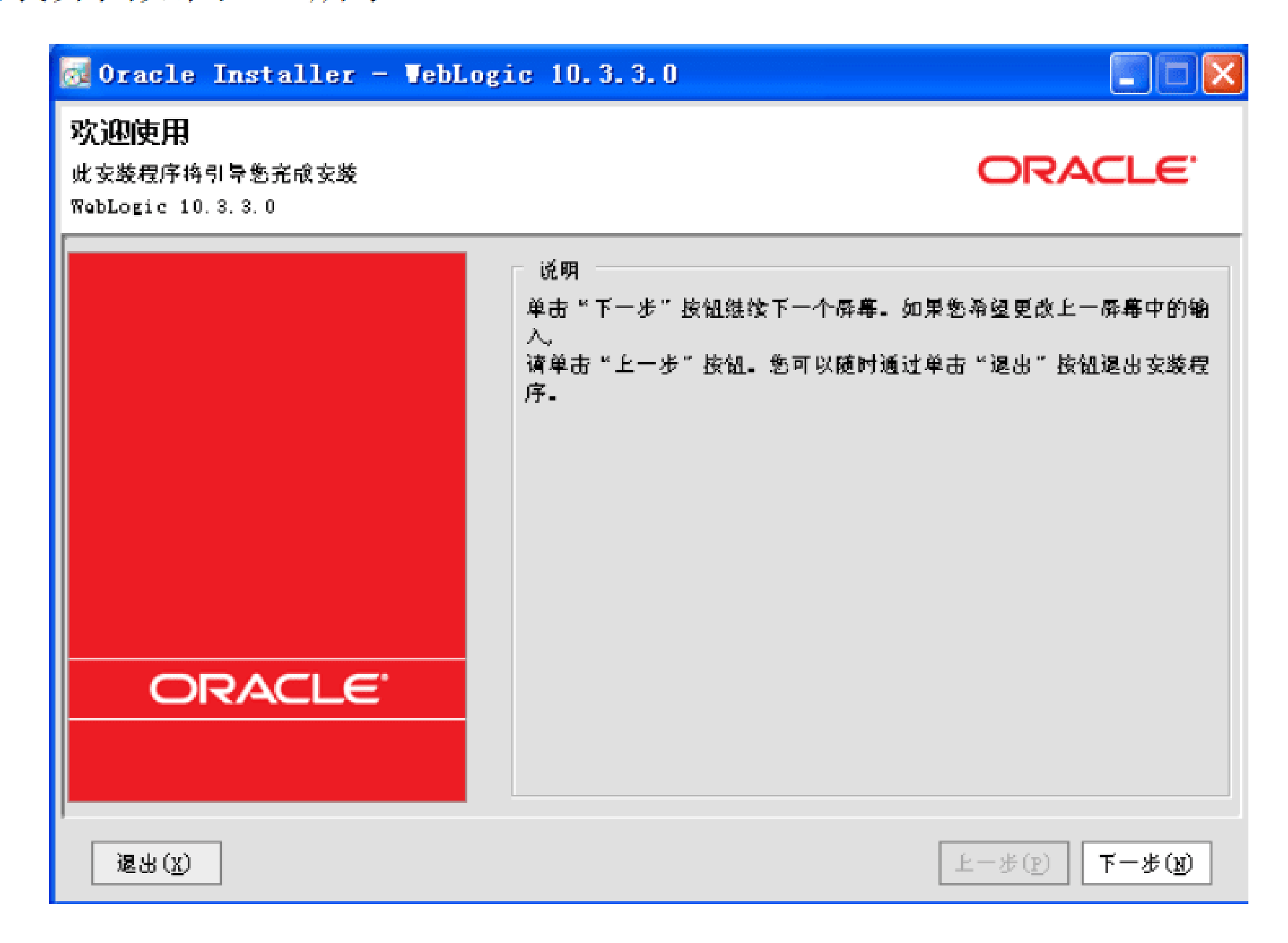


图 2-2

3．选择中间件目录

目录可以修改，建议不要装在其他盘的根目录下，如图 2-3 所示。

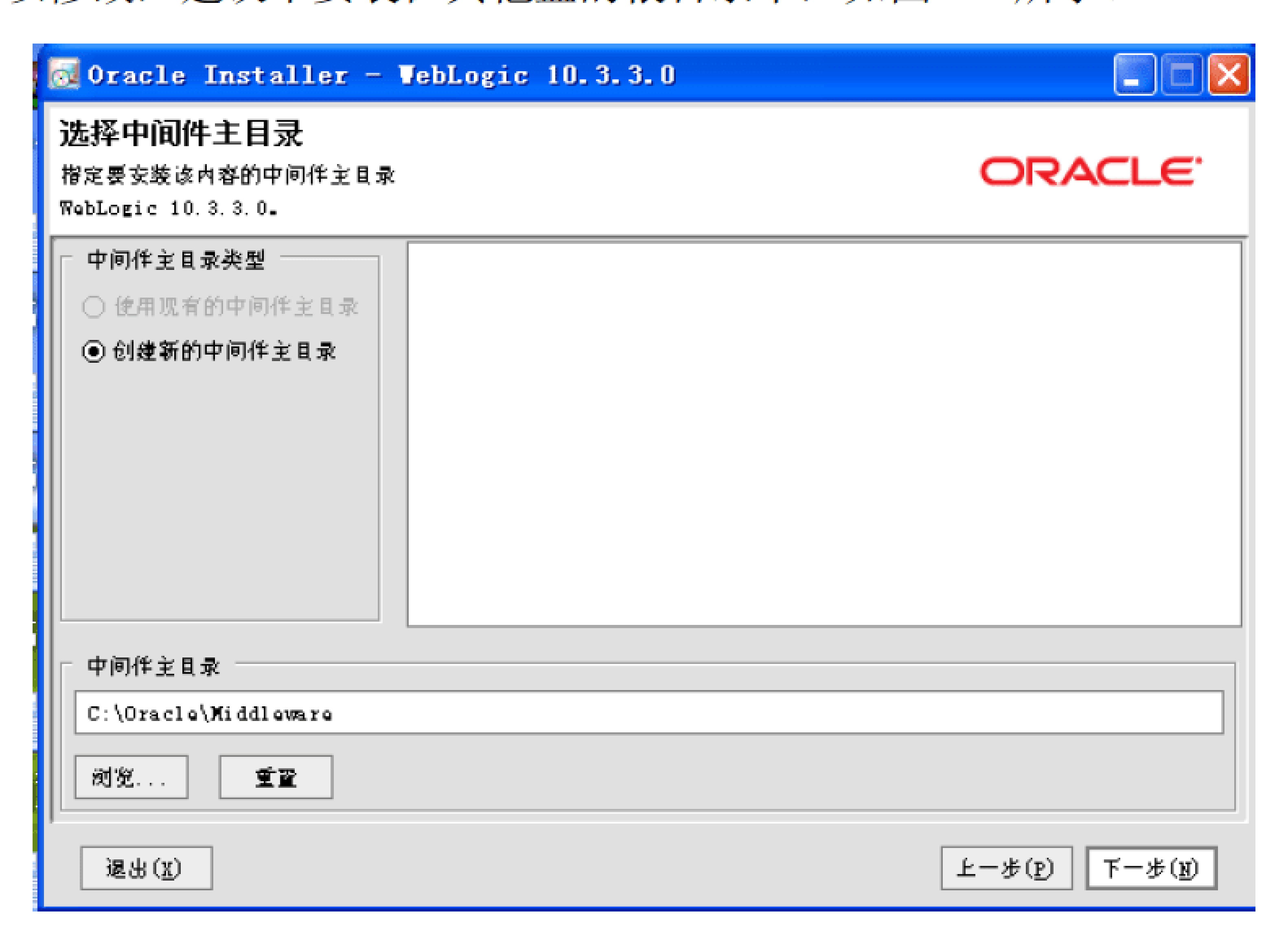


图 2-3

4．注册安全更新

选择暂时不获得安全更新，如图 2-4 所示。



图 2-4

5．选择典型安装

选择安装类型如图 2-5 所示。



图 2-5

6．选择产品安装目录

产品安装目录建议不要修改，如图 2-6 所示。

7．创建快捷方式

创建快捷方式如图 2-7 所示。

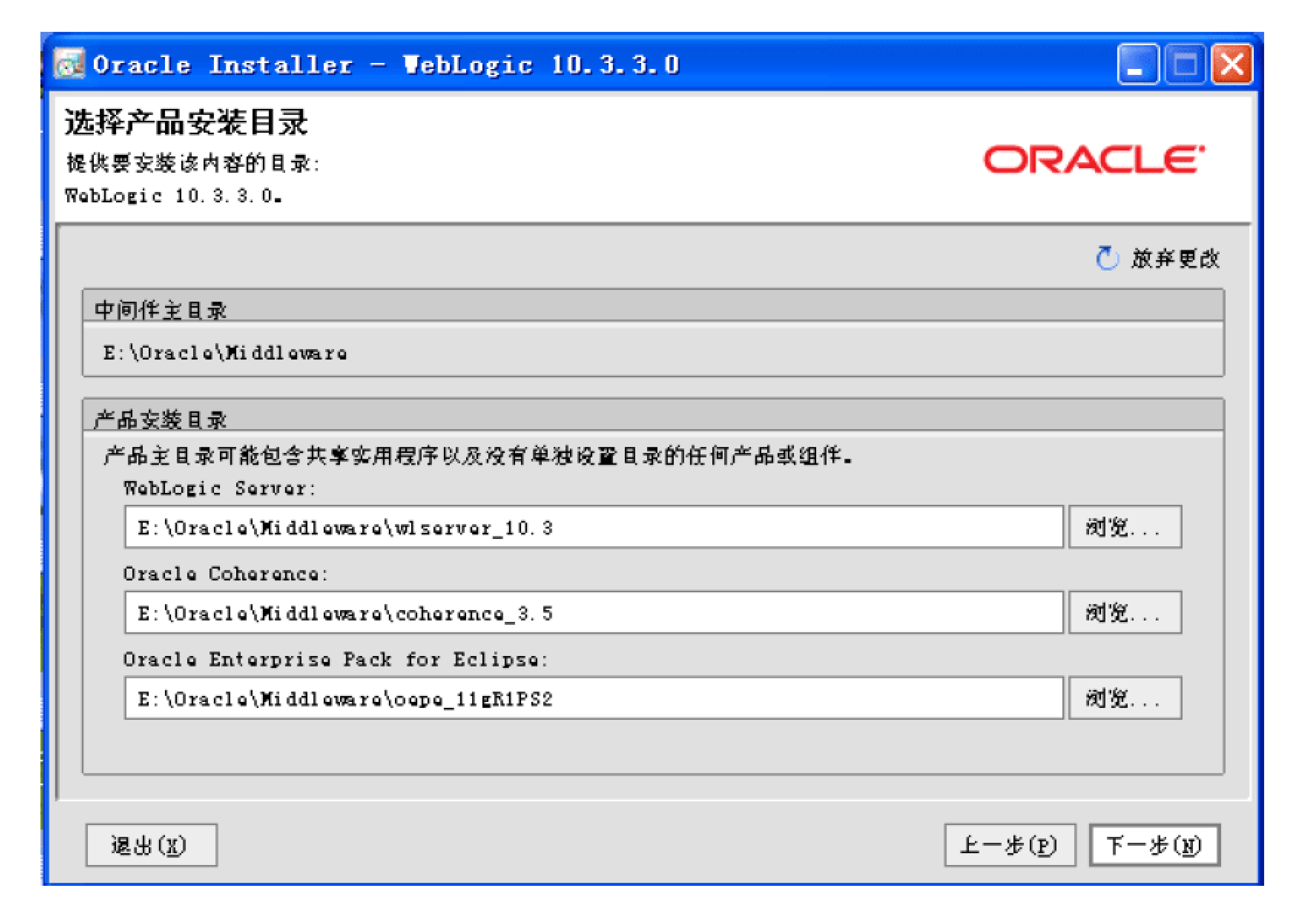


图 2-6



图 2-7

8．显示安装概要

显示安装概要如图 2-8 所示。

9．正在安装

正在安装界面如图 2-9 所示。



图 2-8

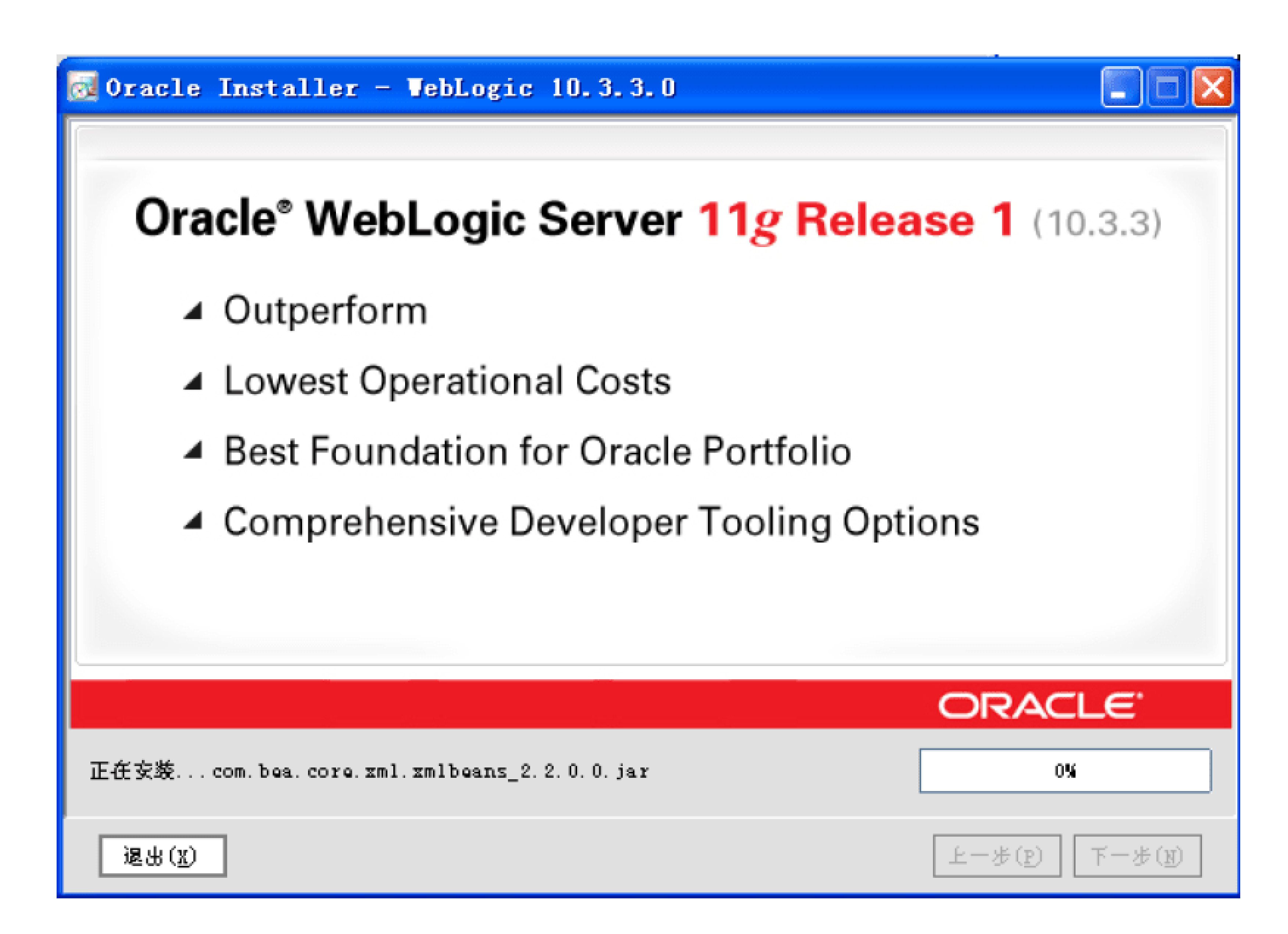


图 2-9

10. 安装完成

完成安装如图 2-10 所示。

至此，一个 WebLogic Server 基本组件安装完成了，但怎么测试组件是否能正常使用呢？答案是创建一个 WebLogic 域，看看服务能否正常启动。

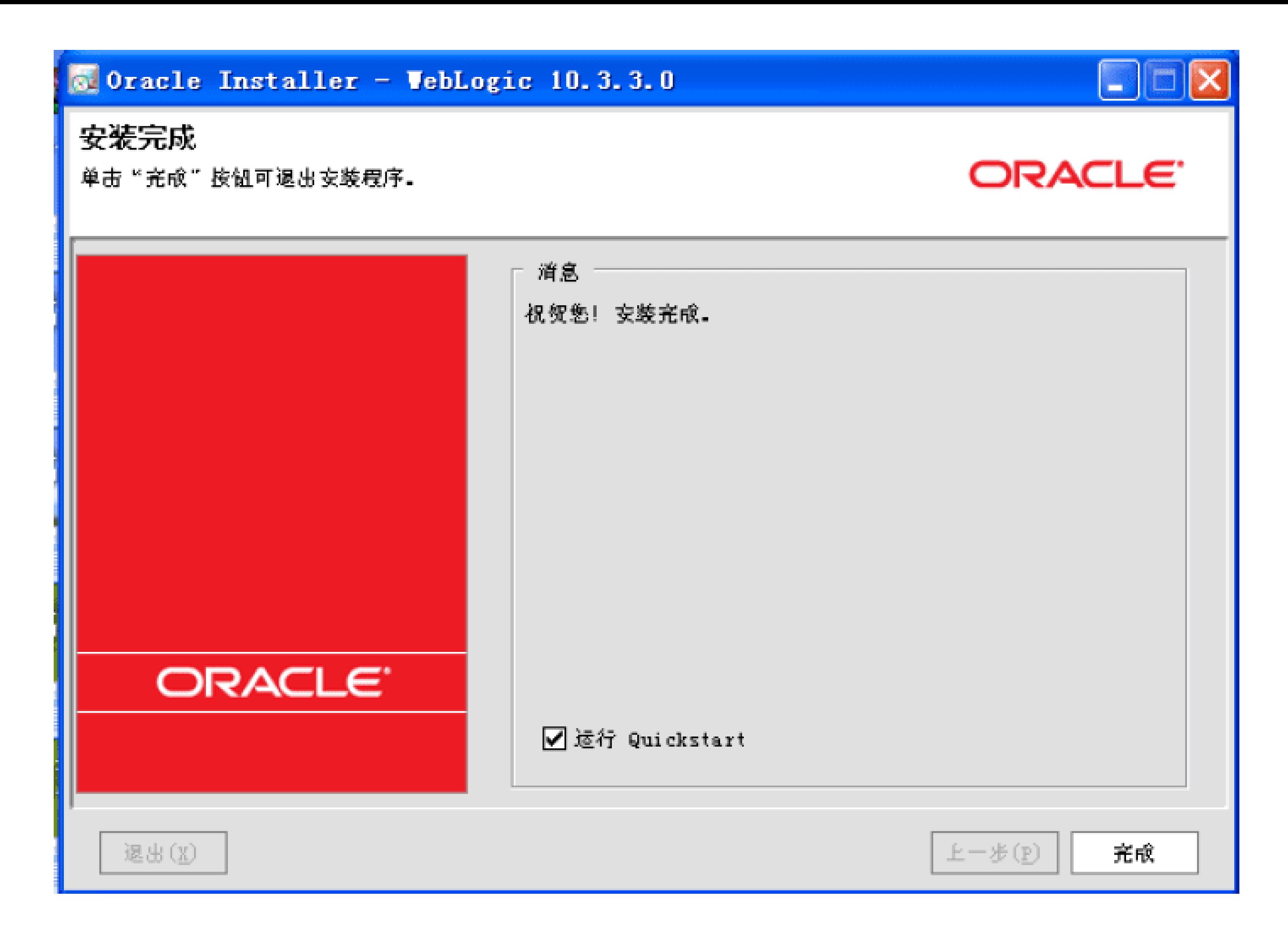


图 2-10

2.3　创建一个 WebLogic 域

1．在完成组件的安装后会默认开启快速启动

单击 Getting started with WebLogic Server®10.3.3 图标，如图 2-11 所示。

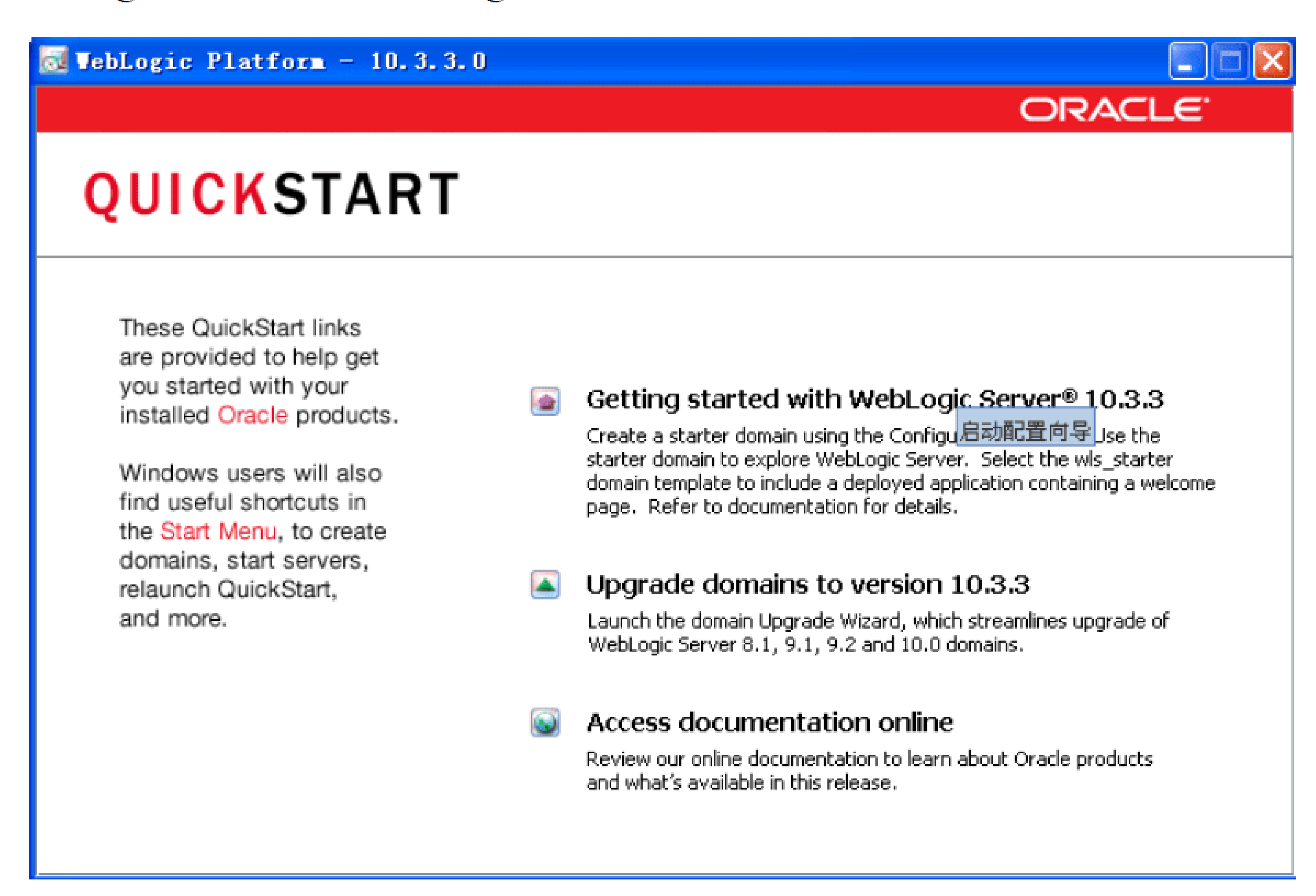


图 2-11

2．进入配置向导

执行“开始”→Oracle Weblogic→WebLogic Server 11gR1→Tools→Configuration Wizard

命令进行创建，进入页面后选择“创建新的 WebLogic 域”单选按钮，单击“下一步”按钮，如图 2-12 所示。



图 2-12

3．选择域源

选择“生成一个自动配置的域以支持下列产品”单选按钮后，单击“下一步”按钮，如图 2-13 所示。



图 2-13

4．指定域名和位置

域名默认为 base_domain，允许更改，域位置建议不要更改，如图 2-14 所示。



图 2-14

5．配置管理员用户名和口令

配置管理员用户名和口令，如图 2-15 所示。

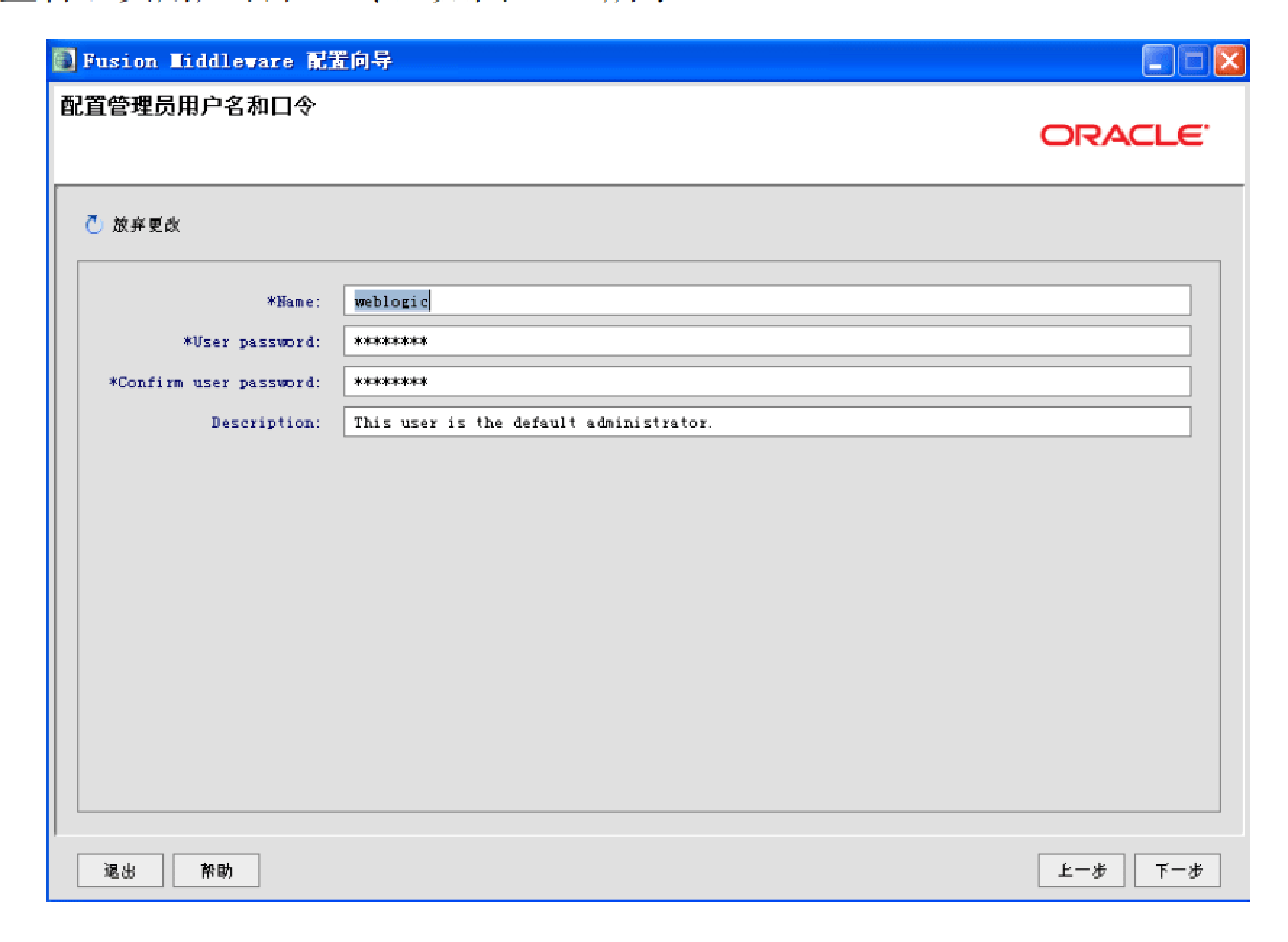


图 2-15

6．配置服务器启动模式

配置服务器启动模式和 JDK，如图 2-16 所示。



图 2-16

7．添加可选配置

选中“管理服务器”复选框并修改其设置，如图 2-17 所示。



图 2-17

8．配置管理服务器

配置管理服务器如图 2-18 所示。

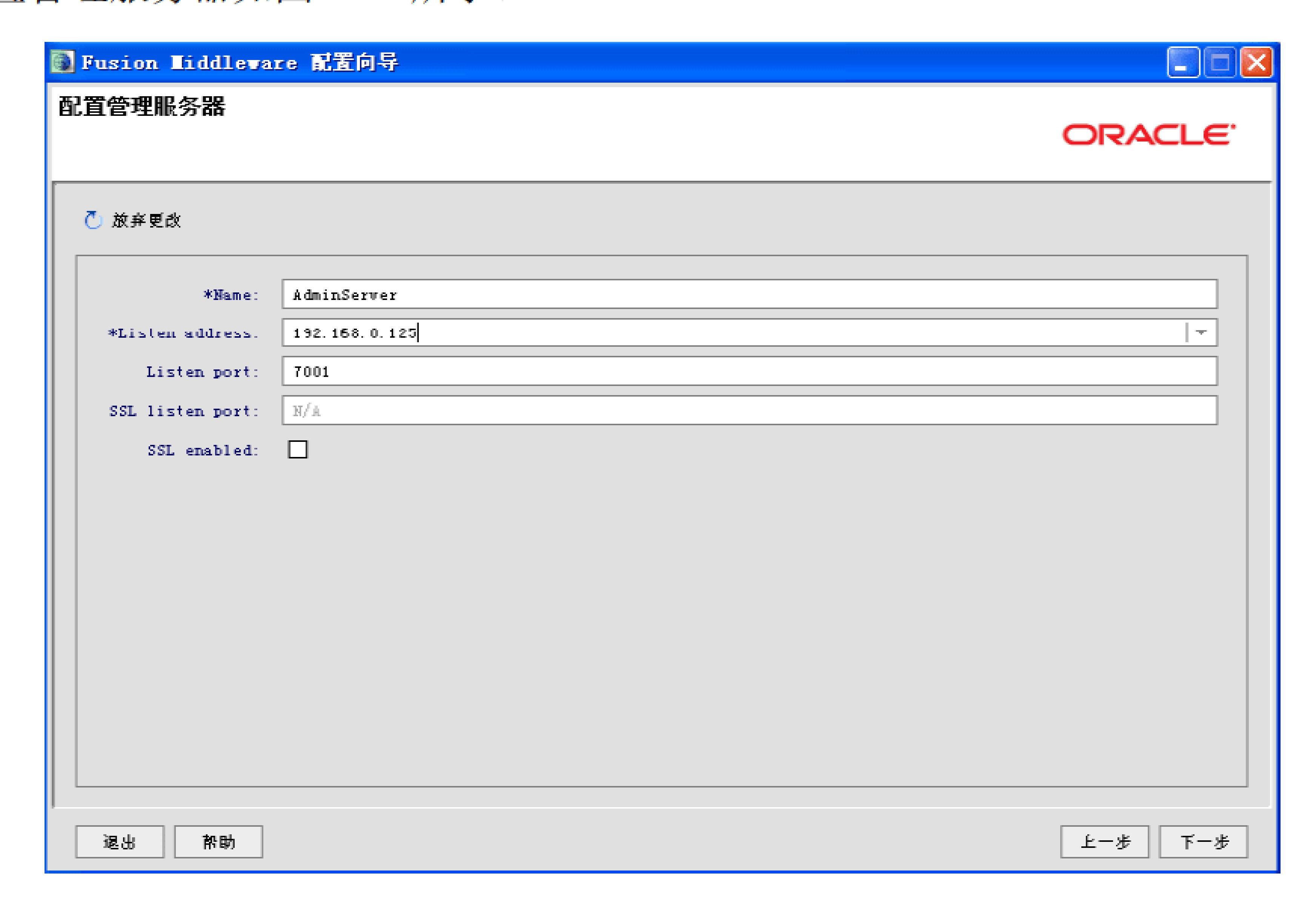


图 2-18

9．显示配置概要

显示配置概要如图 2-19 所示。

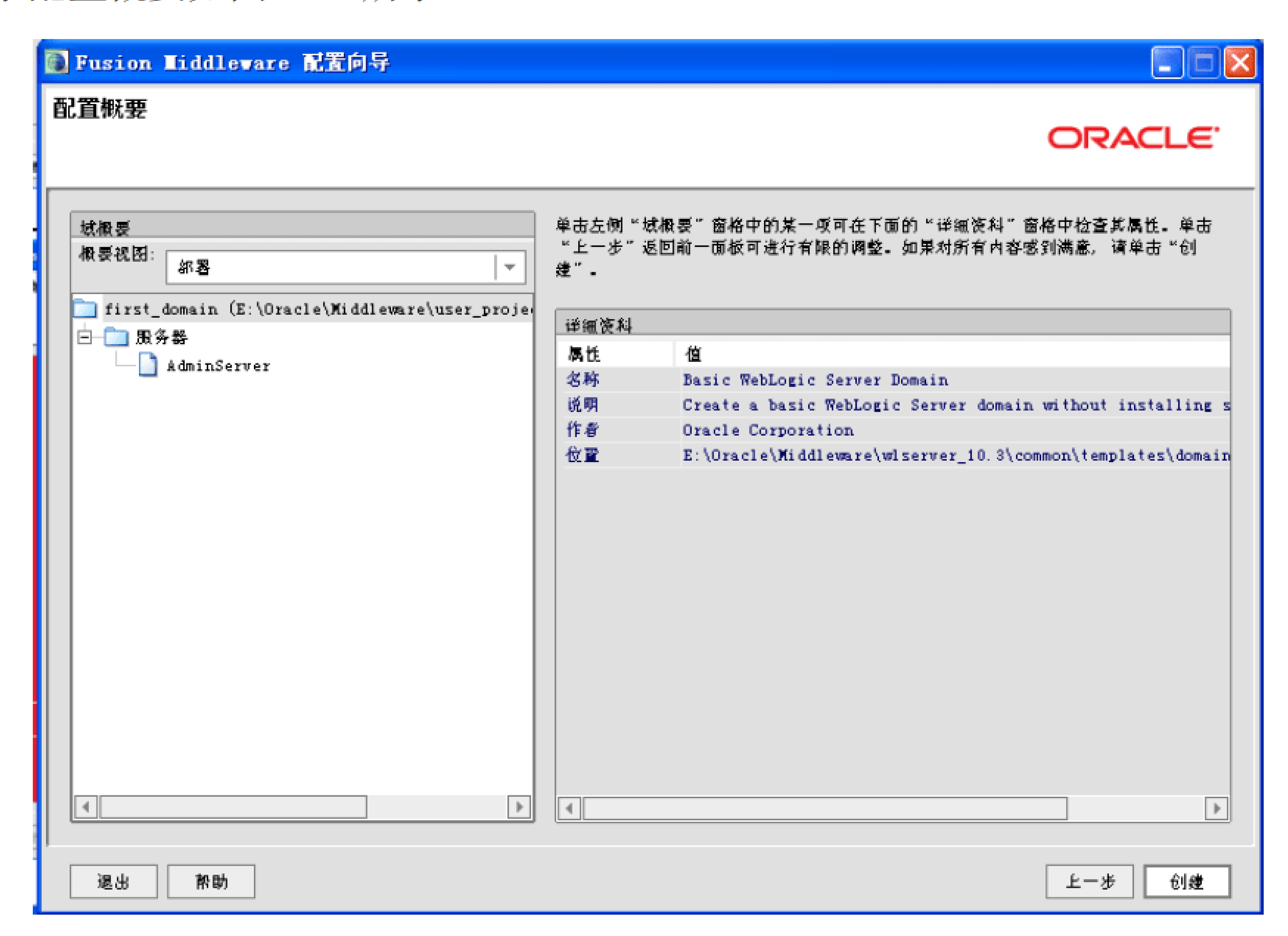


图 2-19

10．正在创建域

创建域如图 2-20 所示。

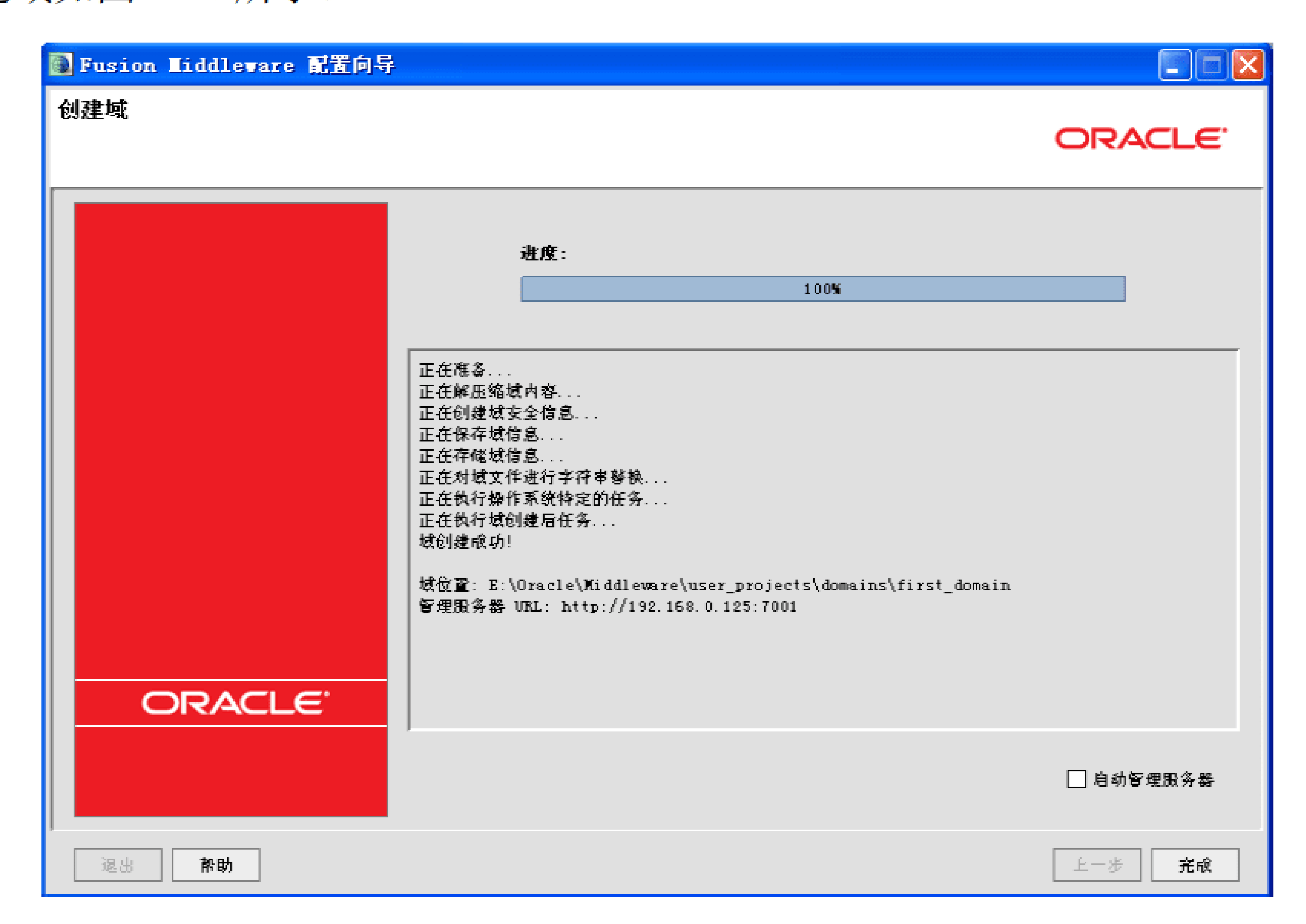


图 2-20

2.4　启动 WebLogic Server

执行“开始”→“程序”→Oracle WebLogic→User Projects→base_domain→Start Admin Server for WebLogic Server Domain 命令，启动 WebLogic Server，如图 2-21 所示。

```
Start Admin Server for Weblogic Server Domain - E:\Oracle\Middleware\...
<2011-3-2 下午01时57分50秒 CST> <Notice> <Security> <BEA-090082> <Security initi
alizing using security realm myrealm.>
<2011-3-2 下午01时58分01秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server s
tate changed to STANDBY>
<2011-3-2 下午01时58分01秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server s
tate changed to STARTING>
<2011-3-2 下午01时58分02秒 CST> <Notice> <Log Management> <BEA-170027> <The Serv
er has established connection with the Domain level Diagnostic Service successfu
lly.>
<2011-3-2 下午01时58分03秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server s
tate changed to ADMIN>
<2011-3-2 下午01时58分03秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server s
tate changed to RESUMING>
<2011-3-2 下午01时58分03秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default
" is now listening on 192.168.0.125:7001 for protocols iiop, t3, ldap, snmp, htt
p.>
<2011-3-2 下午01时58分03秒 CST> <Notice> <WebLogicServer> <BEA-000331> <Started
WebLogic Admin Server "AdminServer" for domain "first_domain" running in Develop
ment Mode>
<2011-3-2 下午01时58分03秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server s
tate changed to RUNNING>
<2011-3-2 下午01时58分03秒 CST> <Notice> <WebLogicServer> <BEA-000360> <Server s
tarted in RUNNING mode>
_
```

图 2-21

可以看见，监听的端口为 7001，监听的 IP 为 192.168.0.125（本机 IP，默认为 127.0.0.1，默认 hostname 为 localhost），服务器的状态为 RUNNING。

2.5　测试安装结果

打开一个浏览器，输入 http://192.168.0.125:7001/console（如果没有对 AdminServer 进行配置，输入 http://localhost:7001/console 即可），进入如图 2-22 所示的页面。

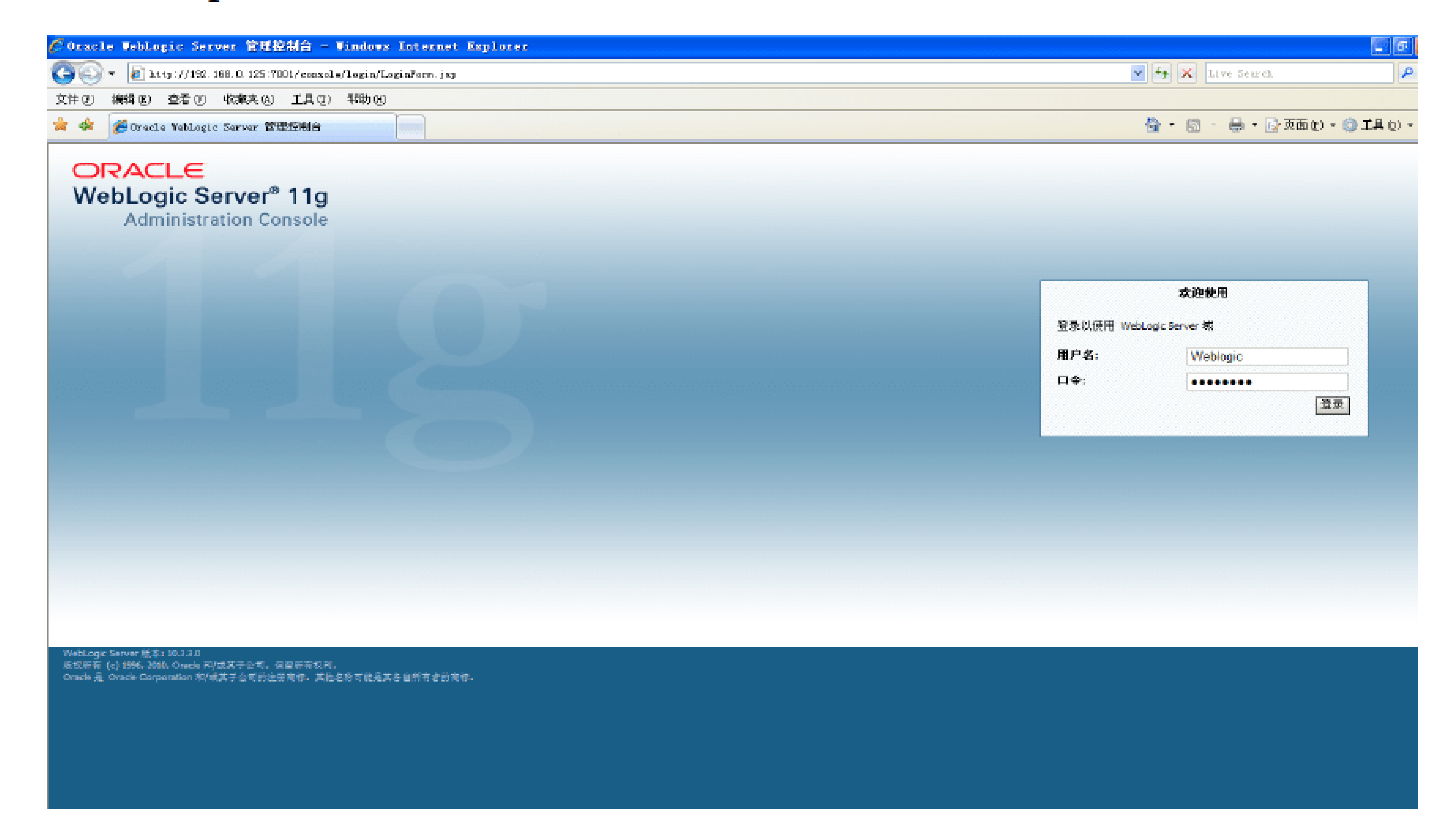


图 2-22

输入您设定的用户名和密码，进入控制台，如图 2-23 所示。

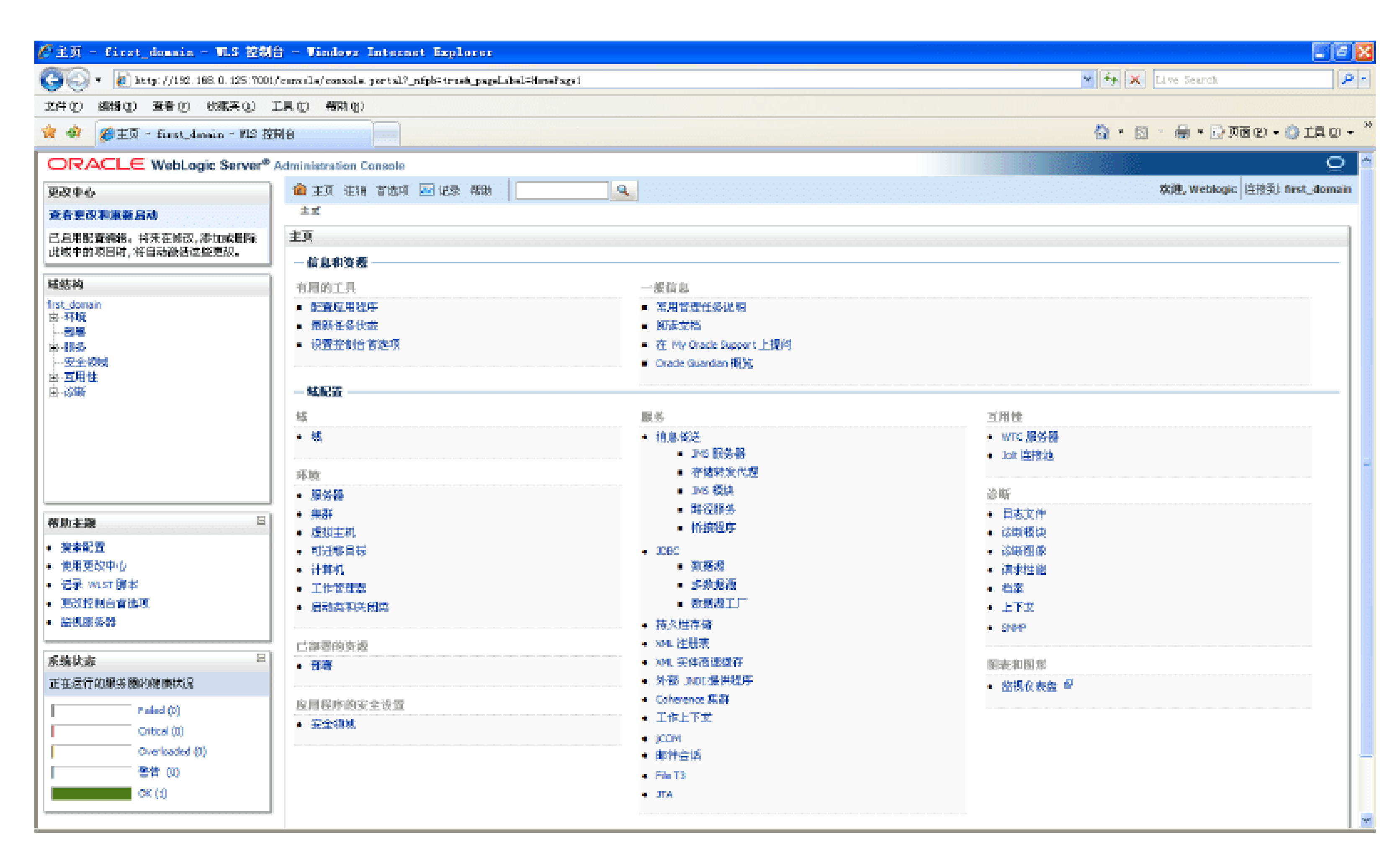


图 2-23

以上说明安装成功。

第 2 篇

基　础　篇

第 3 章　WebLogic 的基本概念

3.1　域 Domain

3.1.1　域的概念、范围和限制

“域”就是逻辑上相关的一组 WebLogic Server 资源，可以作为一个单元进行管理，如图 3-1 所示。

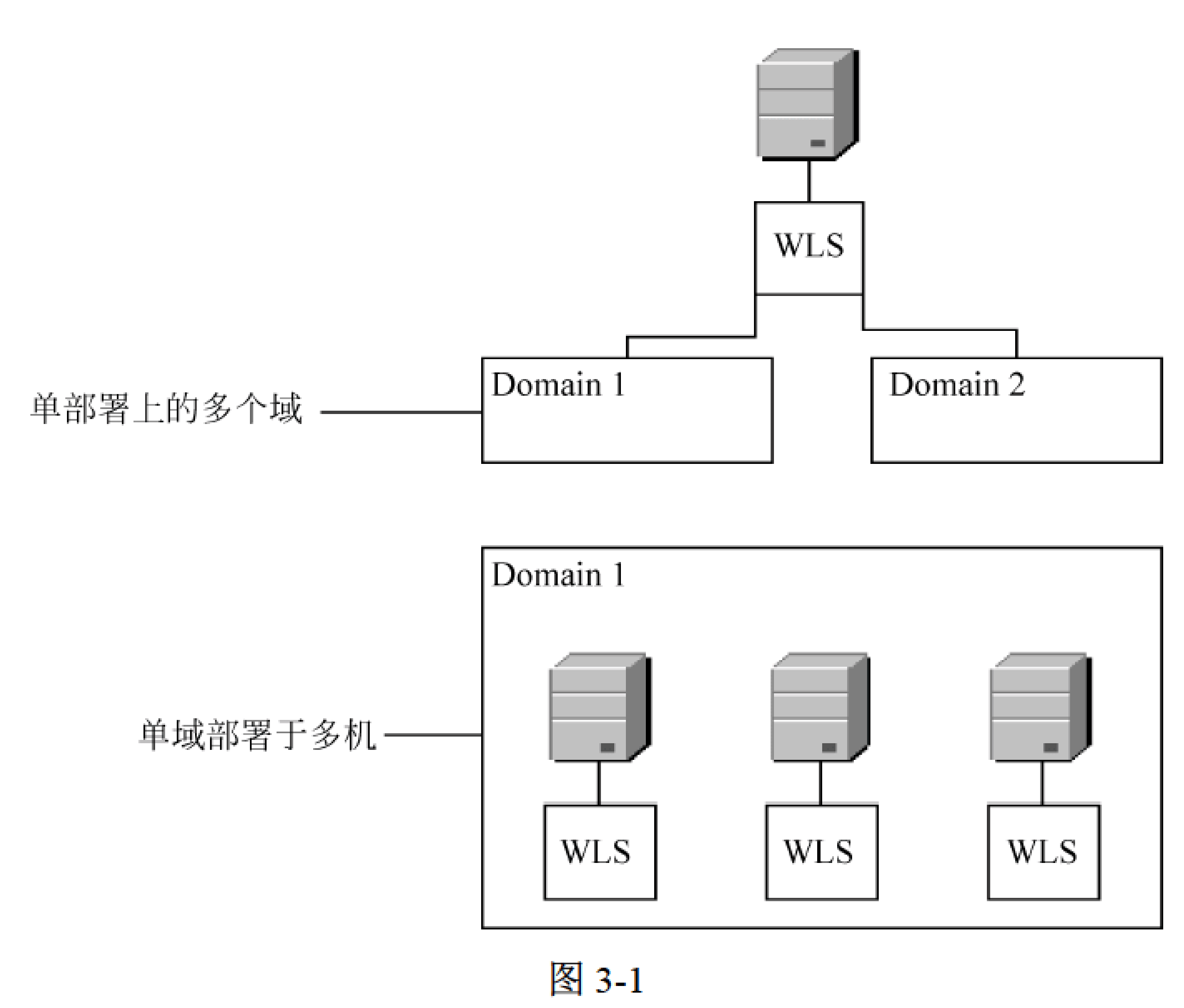


图 3-1

域可以分为以下几种类型。

（1）含有受管服务器的域：简单的生产环境可以只有一个域，该域包含承载应用程序的多个受管服务器和执行管理操作的一个管理服务器。在该配置中，应用程序和资源部署在单个受管服务器上；同样，访问应用程序的客户机也连接至一个受管服务器。

（2）独立服务器域：在开发或者是测试环境中，您可能会在生产域中的服务器之外部署一个独立的应用程序和服务器。可以部署一个简单的域，其中包含一个服务器实例来充当管理服务器并承载您所开发的应用程序。可以随 WebLogic Server 一起安装的 examples 域便是独立服务器域的一个示例。

可以根据以下这些条件组织域。

（1）应用程序的逻辑划分。例如一个程序专用于提供用户最终功能，而另一个专用于后端的财务管理。

（2）物理位置。针对不同的地域或业务分支机构建立域。

（3）大小。以小型单元组织域，有助于更高效地管理域，这些域可能由不同的人员进行管理。

一个域就是一个管理点。

WebLogic Server 可以从逻辑上分区：① 开发应用程序、测试应用程序和生产应用程序；② 组织划分。

在设计你的域配置时，应注意下列限制。

- 每个域要求其履行自己的管理服务器管理活动。当你使用管理控制台来执行管理和监测任务时，可以来回切换域，但在这样做的话，你要连接到不同的管理服务器中。
- 在集群中的所有 Managed Servers 必须位于同一个域中，你不能分割多个域集群。
- 在一个域中的所有 Managed Servers 必须运行 Oracle WebLogic Server 软件的同一版本。

3.1.2　为什么要使用域

WebLogic Server 域是一种管理功能，WebLogic Server 编程不涉及域。所有与域相关的信息都在配置文件中，只有管理员需要考虑域。

域是一种管理功能，具有以下特点。

（1）对应用程序是透明的。

（2）可以根据技术或者是业务需要对其进行配置和管理，即使是在部署应用程序之后或在生产使用中。

（3）WebLogic Server 域可用于区分：① 开发应用程序、测试应用程序和生产应用程序；② 管理和运营职责；③ 组织或业务划分。

（4）域的多个优势如下。

一个企业可能有多种不同的应用程序，他们在地理上可能是分散，也可能被组织到不同的职责领域中，因此可能有多个独立的域。

每个域都是一个单独的管理单元。可以根据地理因素（位于同一位置的所有计算机）对其进行组织，也可以根据企业中的部门进行划分（财务、制造、运输）对其进行组织。

最后，企业可能要求其多个域中的应用程序能够进行互操作。通常可能的做法是将一个域扩展以包容整个企业，但扩展的域会因计算机和服务的数量众多而变的无法管理。因为一个域必须作为一个管理单元来管理，所以配置将会快速变的非常庞大，管理难度要远大于开发和实现应用程序。

要保持域相对紧凑，易于管理，必须有一种方法将应用程序划分到多个域，并且使一个域中的应用程序仍然可以访问其他域中的服务，WebLogic Server 域功能可以实现域之间的通信。

3.2　服务器

3.2.1　管理服务器

服务器是一个在 JVM 中执行的 weblogic.server 实例。

WebLogic Server 只是在执行一个 weblogic.server 类的 Java 虚拟机。

weblogic.server 类包含有 main()方法，用于启动 WebLogic 系统。由于服务器在单个 Java 虚拟机中执行，它占用专用的 RAM 容量，该容量是在系统引导时确定的。一个服务器仅可以与一台计算机相关联，但一台计算机可以与多台服务器相关联。

管理服务器是域的中央控制点，存储域的配置信息和日志运行 WebLogic 管理控制台，如图 3-2 所示。

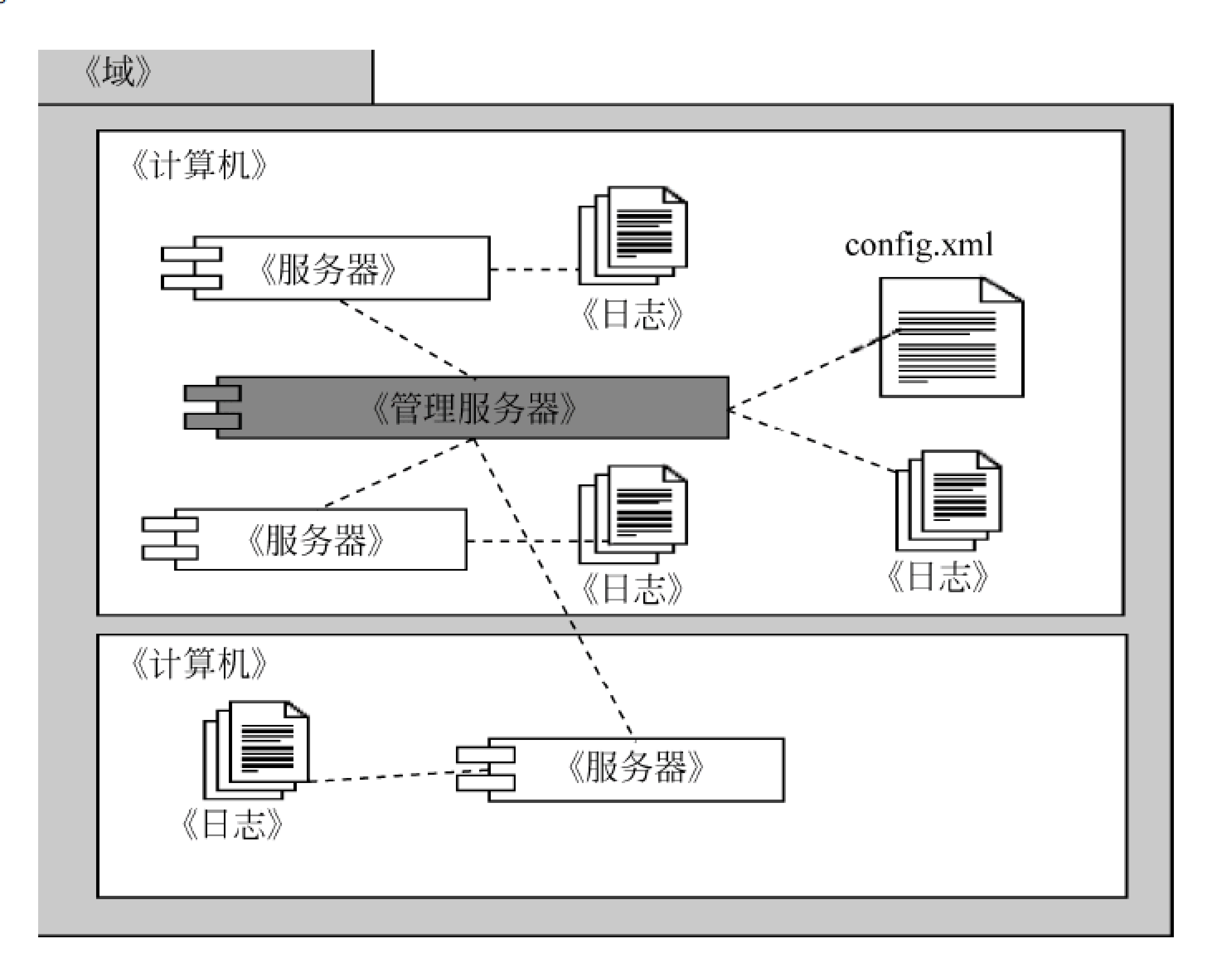


图 3-2

管理服务器是用做配置整个域的中央控制实体，可以通过以下方式调用管理服务器的服务。

（1）WebLogic Server 管理控制台：管理控制台是用来配置域的基于浏览器的图形界面（GUI）

（2）WebLogic Server 应用程序编程接口（API）：可以使用 WebLogic Server 提供的 API 编写 Java 类来修改配置特性。

（3）WebLogic Server 命令行实用工具（weblogic.Admin）创建脚本，实现自动化的域管理。

（4）SNMP：可以实现简单的网络管理协议来监视 WebLogic Server 域。

要修改域配置，域管理服务器必须正在运行。管理服务器是通过写 config.xml 文件来维护域的配置信息的。

3.2.2 受管服务器

受管服务器指域中任何不属于管理服务器的服务器，与管理服务器联系以获得配置信息，在生产环境中运行业务程序，如图 3-3 所示。

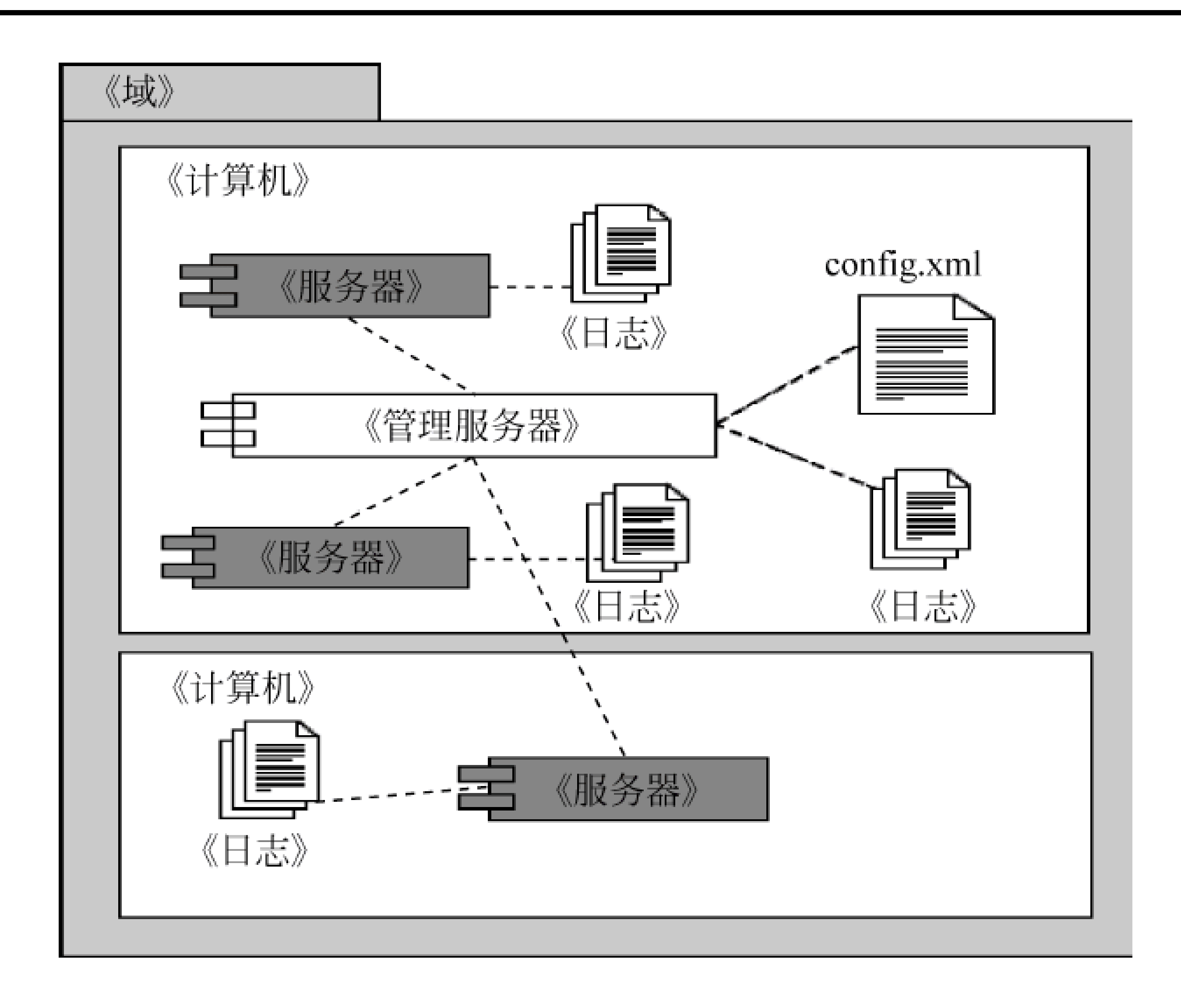


图 3-3

受管服务器是一个 WebLogic Server 实例，它从管理服务器中检索域配置数据。域中可以有多个受管服务器，但是只有一个管理服务器。通常，在创建并启动作为受管服务器的服务器实例时，在生产环境中要运行业务应用程序。在该标准场景中，作为管理服务器启动的服务器实例不会运行业务应用程序，它仅管理域中的资源。为提高可靠性和性能，用户可以在多台计算机上安装 WebLogic Server 软件，并在不同的 WebLogic Server 主机上运行已创建的服务器。

3.3　计算机 Machine

3.3.1　Machine 的概念

Machine 是承载一个或多个 WebLogic Server 的计算机，运行受支持的操作系统平台并且可以承载多个 WebLogic Server 实例，如图 3-4 所示。

计算机表示承载一个或多个 WebLogic Server 实例的物理计算机。

WebLogic Server 使用所配置的计算机名来确定将任务（如 HTTP 会话复制）委托给集群中的哪个服务器是最佳的。管理服务器使用此计算机定义与节点管理器应用程序来启动远程 WebLogic Server 实例。

3.3.2　为什么要使用 Machine

（1）可以对应到服务器所在的物理硬件。

（2）可以用来远程管理和监控。

（3）用于加强 fail over 管理。

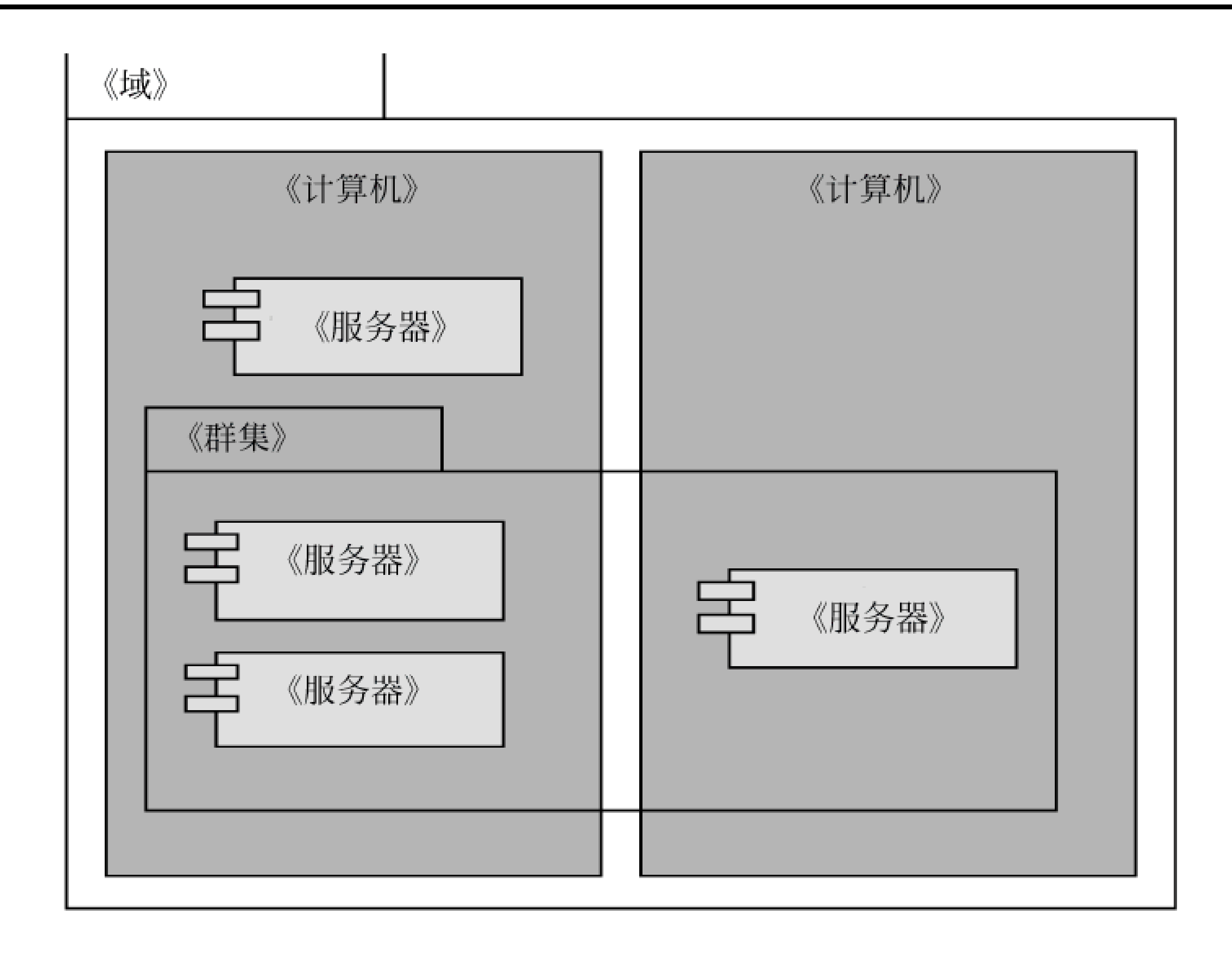


图 3-4

3.4 集群 Cluster

3.4.1 集群的概念

1．什么是集群

集群就是一组相互协作的 WebLogic Server 实例。

2．集群的优点

将 WebLogic Server 集群在一起有两个主要优点。

（1）可伸缩性：是为应用程序提供更多容量的能力，在这里，是通过添加更多的服务器而无需进行重大的架构更改实现的。

（2）高可用性：确保当（集群中的）某个服务器出现故障时，有其他服务器接管其工作，从而不会对客户端造成影响。

3.4.2 主要功能

WebLogic 集群的主要功能如下。

（1）应用程序故障转移：应用程序中正在执行任务的某个对象不可用时，另一个对象将接管并完成其工作。

（2）站点故障转移：一个站点的所有服务器和应用程序出现故障时，这些服务和当某个服务器出现故障时，其上的固定服务可迁移到集群中的另一个服务器。

（3）负载平衡：多个服务器中均匀地分配任务和通信。

3.4.3　基本集群架构

基本集群架构将静态 HTTP、呈现逻辑、业务逻辑和对象组合到一个集群中（图 3-5）。

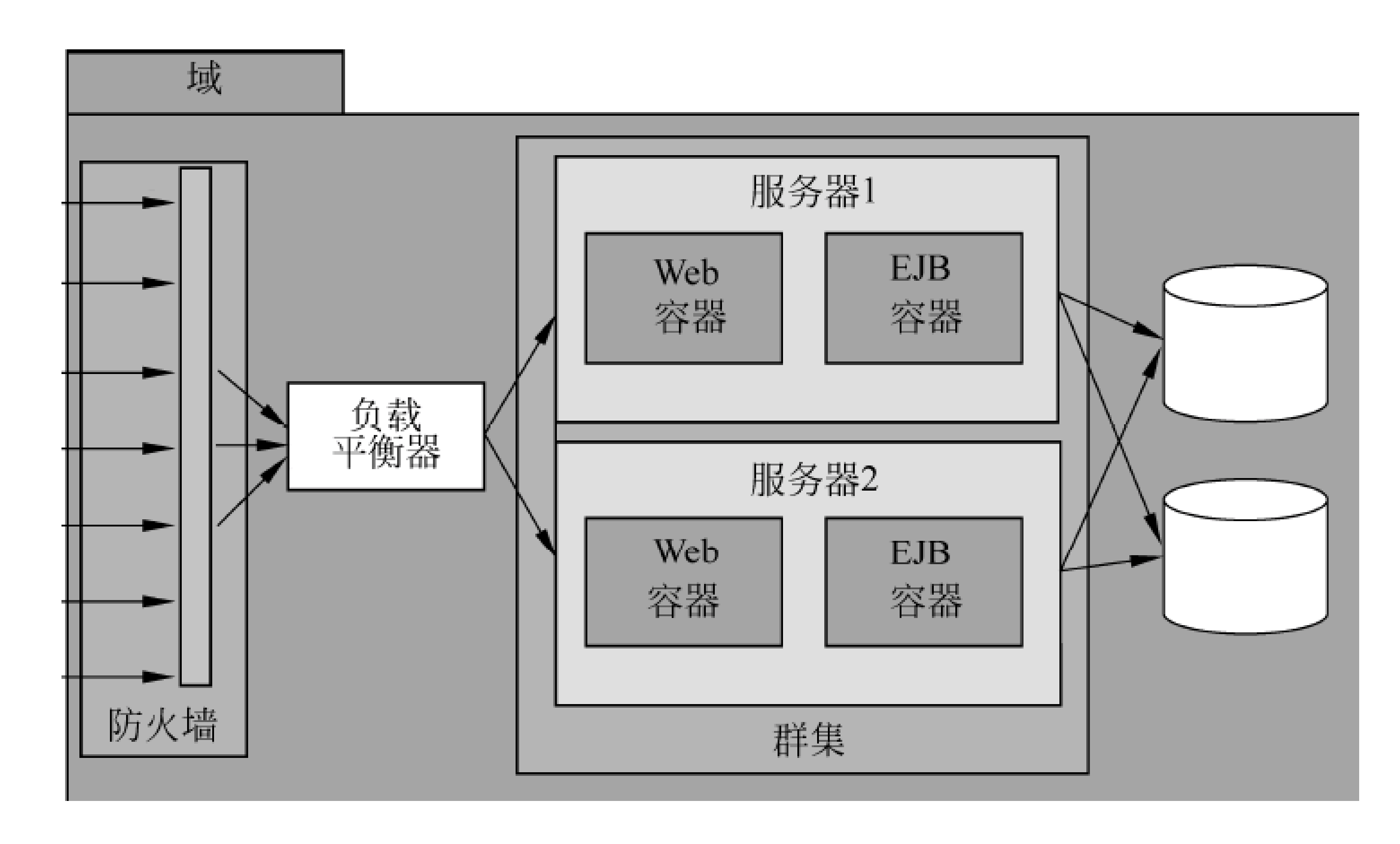


图 3-5

3.4.4　多层集群架构

Web 层和提供服务的业务逻辑可分别放到两个集群中，如图 3-6 所示。

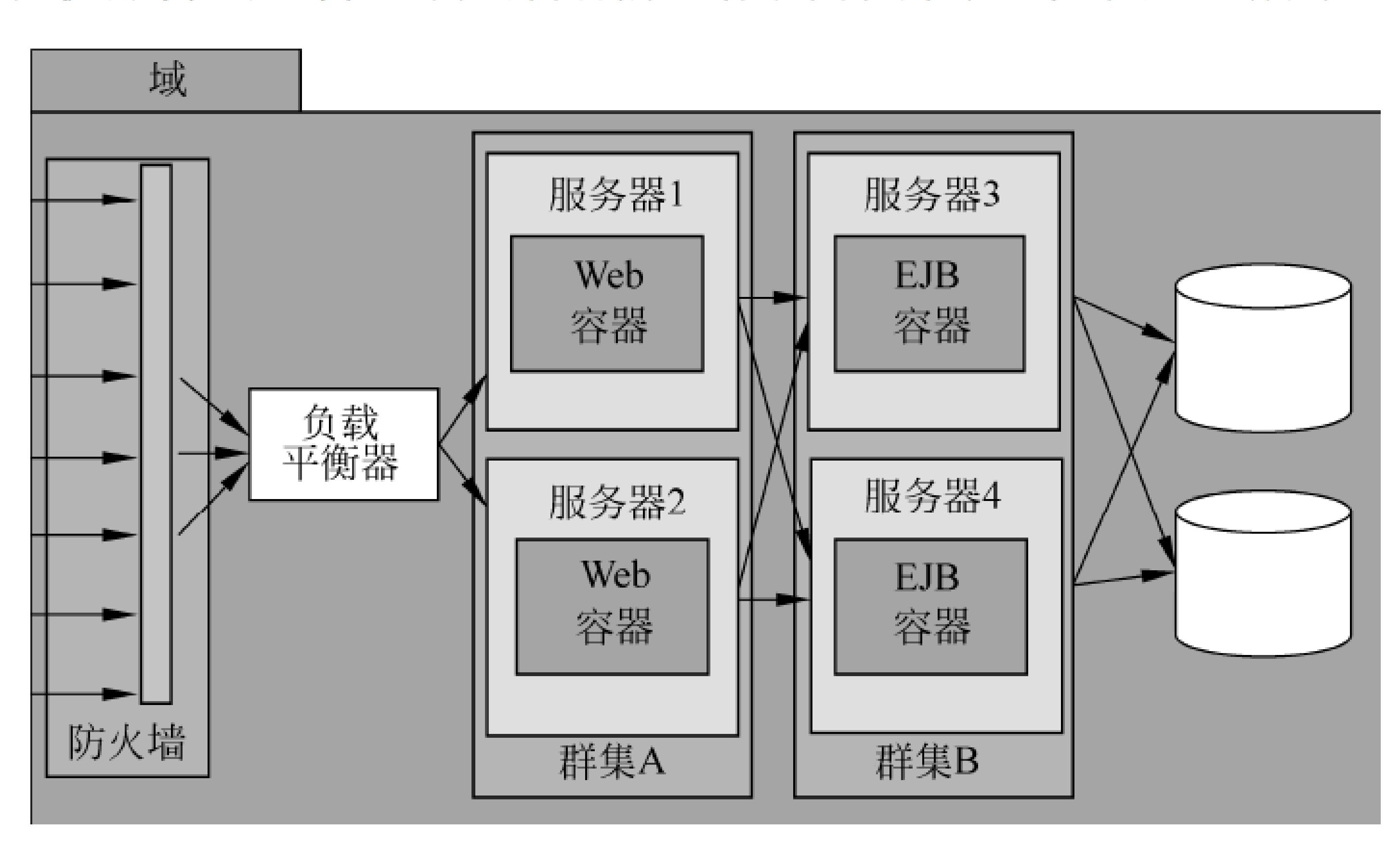


图 3-6

建议多层架构使用两个单独的 WebLogic Server 集群：一个提供静态 HTTP 内容和集群 Servlet，另一个提供集群 EJB。

3.4.5　何时使用多层集群架构

建议对有以下要求的 Web 应用程序使用多层集群。

（1）需要对集群 EJB 的方法调用进行负载平衡。
（2）需要在提供 HTTP 内同时提供集群对象的服务器之间进行灵活的负载平衡。
（3）需要更高的可用性（减少单点故障数）。
（4）更灵活的安全性规划。

3.4.6 基本集群架构的优缺点

基本集群架构具有以下优点。
（1）易于管理。
（2）灵活的负载平衡。
（3）可靠的安全性。
基本集群架构具有以下缺点。
（1）无法对 EJB 方法进行调用实现负载平衡。
（2）跨层平衡负载可能会出现不平衡。

3.4.7 多层集群架构的优缺点

多层集群架构具有以下优点。
（1）负载平衡得到改进。
（2）可对 EJB 方法进行负载平衡。通过分别在不同的集群上承载 Servlet 和 EJB，Servlet 中对 EJB 的方法调用可以在多个服务器间进行负载平衡。
（3）更高的可用性。通过使用更多 WebLogic Server 实例，多层架构的故障点要比基本集群故障点少。
（4）安全方案得到改进。将呈现层和对象层放到不同的集群上，可以使用只在 DMZ 中放置 Servlet/JSP 集群的防火墙策略。通过拒绝来自不可信客户端的直接访问，承载集群对象的服务器可以得到进一步的保护。
多层集群架构具有以下缺点。
（1）呈现层频繁调用业务逻辑时会造成瓶颈。
（2）许可证成本增加。
（3）增加了防火墙配置的复杂性。

3.4.8 代理服务器

代理服务器用于为集群提供负载平衡和故障转移。此外，此类服务器还具有以下特点。
（1）是客户端与集群交互的第一级。
（2）使集群看上去就像是一个服务器。
代理服务器可以基于软件也可以基于硬件。基于软件的代理服务器可以是 WebLogic 提供的第三方 Web 服务器插件和 Proxy Servlet 或第三方应用程序。基于硬件的代理服务器通常是物理负载平衡器（如 Local Director 或 F5 Networks Big IP）。

3.4.9　基本集群代理架构

除静态内容由非集群 HTTP 服务器承载以外，其余都与基本集群架构类似（图 3-7）。

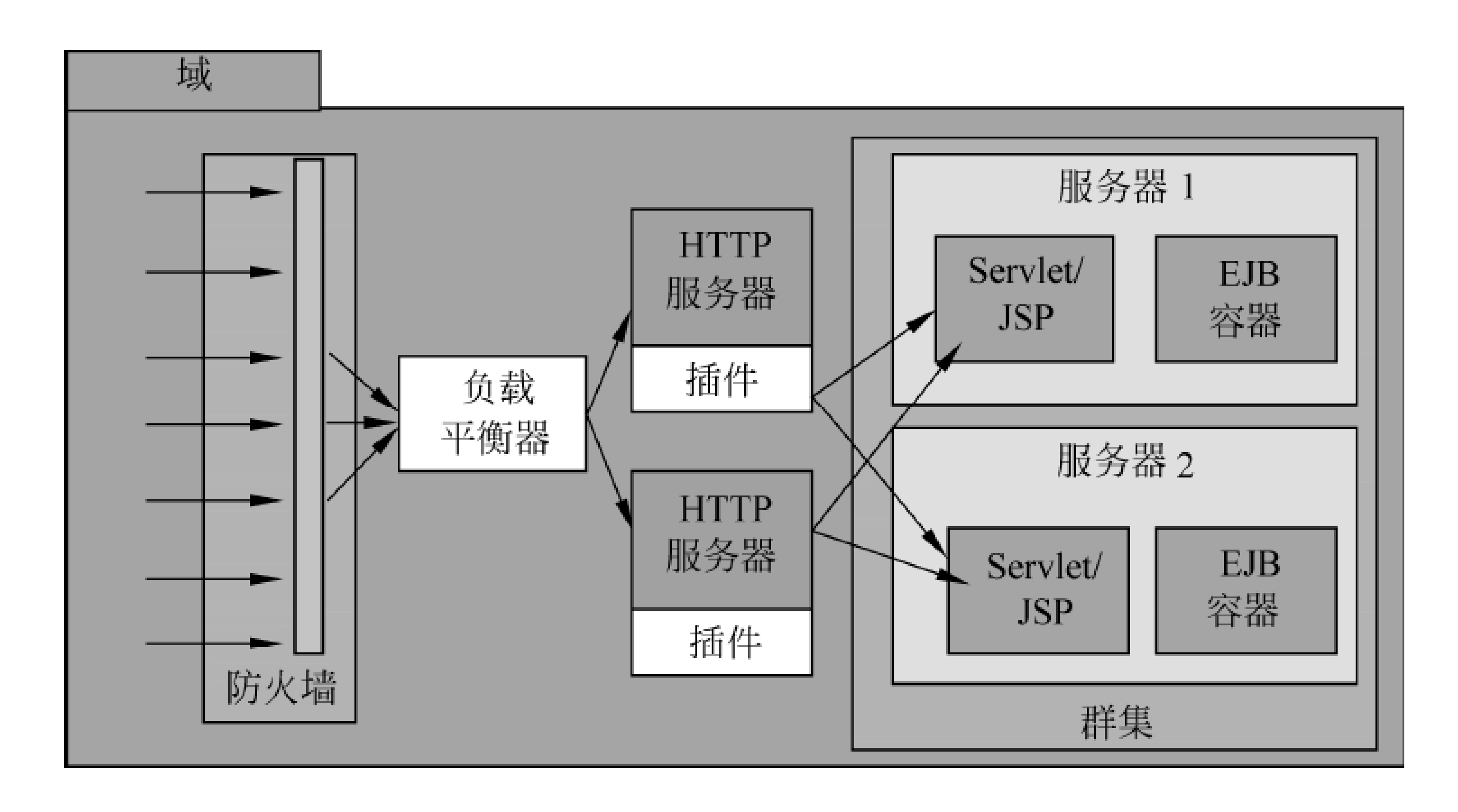


图 3-7

带有外部 Web 服务器的基本集群架构包含两个物理硬件和软件层。

该架构利用硬件和软件层来专门执行提供应用程序 Web 层的任务，该物理 Web 层由一台或多台相同配置的计算机组成，这些计算机承载下面一种应用程序组合。

（1）带有 HttpClusterServlet 的 WebLogic Server。

（2）带有 WebLogic Server Apache 插件的 Apache。

（3）带有 WebLogic Server NSAPI 代理插件的 Netscape Enterprise Server。

（4）带有 WebLogic Server Microsoft IIS 插件的 Microsoft Internet Information Server。

3.4.10　多层集群代理架构

除静态内容由非集群 HTTP 服务器承载以外，其余都与多层集群架构类似，如图 3-8 所示。

使用独立的 Web 服务器和代理插件具有以下优势。

（1）可以使用现有硬件。

（2）如果已有向客户端提供静态 HTTP 内容的 Web 应用程序架构，则可以轻松地将现有 Web 服务器与一个或多个 WebLogic Server 集群集成，以提供动态 HTTP 和集群对象。

（3）可以使用熟悉的防火墙策略。

在 Web 应用程序前端使用 Web 服务器代理，可以使用熟悉的防火墙策略来定义 DMZ。通常，当不允许直接连接到架构中的其余 WebLogic Server 集群时，您还可以继续在 DMZ 中放置 Web 服务器，如图 3-8 所示。

但是该方案也存在一些缺点。

（1）管理成本增加。

（2）负载平衡方案受到限制。

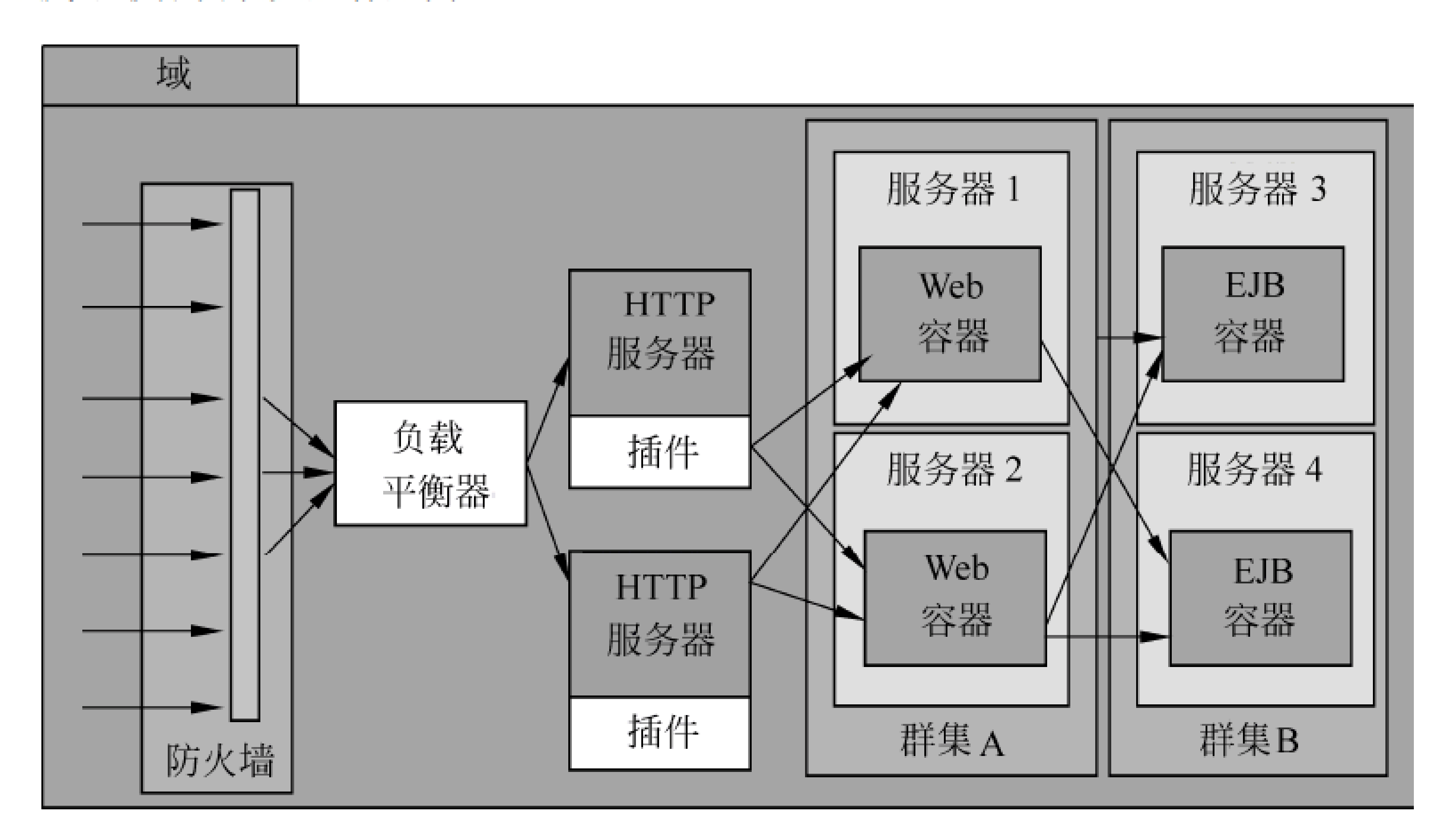


图 3-8

3.5 开发模式与生产模式

（1）开发模式，该模式用于启用自动部署。

（2）生产模式，该模式用于关闭自动部署。

WLS 支持 3 种部署方法：自动部署、控制台部署、命令行部署。

自动部署是一个标志，它使得配置或定位工作不需要任何管理员或开发人员的操作。当其处于启用状态时，管理服务器将监视指定的目录以确定是否有新建的或更新的应用程序。WebLogic Server 将自动部署该目录中的所有文件或目录。WebLogic Server 配置为定期监视该目录以确定新建的或更新的文件。

默认情况下，自动部署目录为域的\autodeploy 子目录。在 autodeploy 目录中找到的任何新文件都将被自动部署。applications 目录中编辑或修改过的归档文件将会重新部署。如果应用程序处于展开格式，则创建一个名为 REDEPLOY 的空文件，将其放置在 WEB-INF 目录中，然后当应用程序需要重新部署时，修改该文件以更新其时间戳。服务器将定期检查对 REDPLOY 文件时间戳的更新。

最后，当前存在于 applications 目录中，但后来被删除的所有文件都将自动从应用服务器中取消部署。

第 4 章　WebLogic 主要目录结构

4.1　总体目录结构分布

|-bea　　BEA 的主目录

|-/ jdk_xxx　　包含 Sun JDK 软件（如果此软件已随您的软件进行了安装）。JDK 提供了 Java 运行时环境（JRE）和工具，用于对 Java 应用程序进行编译和调试。在此目录名称中，xxx 表示您系统上安装的 Sun JDK 版本，例如 jdk160_05

|-/jrockit_xxx　　包含随您的软件安装的 BEA JRockit JDK 软件。JDK 提供了 Java 运行时（JRE）和工具，用于对 Java 应用程序进行编译和调试。在此目录名称中，xxx 表示您系统上安装的 JRockit JDK 版本，例如 jrockit_160_05

|-/logs 目录　　包含 BEA 主目录的安装和卸载历史记录文件

|-/user_projects 目录　　包含着用户创建的域信息，在 Domains 下面分开存放着域，域下面存放着各自的应用程序

|-/util 目录　　包含用于至此此 BEA 主目录中安装的所有 BEA 产品安装的实用工具（第三方工具）。Utils.jar 文件包含支持 UpdateLicense 实用工具的代码

|-/ wlserver_10.3　　WebLogic Server 主目录，包含 WebLogic 安装的所有组件信息

|-/ workshop_10.3　　WebLogic 工作车间，包含 WebLogic 开发用的工具、包、可执行文件等

|-/tools 目录　　eclipse 工具不是 IDE 版本

|-/modules 目录 　　系统及应用运行时所需的 jar 文件

|-/registry.xml 文件 　　一个注册表文件，包含目标系统上安装所有 BEA 产品的持久性记录。此注册表包含与产品相关的信息，例如版本号、Service Pack 编号以及安装目录的位置。注意：请勿手工编写此文件。这样做可能会导致当前安装的 BEA 产品出现运行问题，或者导致在稍后为维护升级而安装 BEA 产品时出现问题

4.2 user_projects 目录

域目录结构（user_projects 目录包含着用户创建的域信息，在 Domains 下面分开存放着域，域下面存放着各自的应用程序，现以 domains\base_domain 为例，对其下的目录和部分文件结构进行说明）。

|-/autodeploy 　　当 WebLogic 服务实例以开发模式进行时，此目录下的应用程序（ear,war 等）将被自动展开到服务

|-/bin 　　包含启动配置教本，cmd 为 Windows 下运行教本，sh 为 Linux 下运行教本。如下设置域环境：setDomainEnv.sh 启动管理控制器，如果有多个 startManagedWebLogic.sh 启动节点控制器，startPointBaseConsole.sh 启动服务，startWebLogic.sh 停止管理控制，stopManagedWebLogic.sh 停止服务，stopWebLogic.sh。注意：startWebLogic.sh 启动的是域管理器，具体起哪个应用由配置文件决定

|-/config 　　本域相关的配置文件都在这里，如 JMS、JDBC 等

|-/console-ext 　　保存扩张控制台信息，只应用于 admin server

|-/init-info 　　启动域管理器的初始化配置文件目录

|-/lib 　　域库目录

|-/security 　　安全相关

```
|-/servers                   包含域的所有服务

|-/user_staged_config        可选

|-/fileRealm.properties

|-/startWebLogic.cmd         调用 bin\startWebLogic.cmd 文件启动服务

|-/startWebLogic.sh          调用 bin\startWebLogic.sh 文件启动服务
```

4.3　utils 目录

```
|-/bsu 目录                  包含可执行文件 bus.jar 和客户端补丁 jar 文件
|-/quickstart 目录           快速学习目录
|-/uninstall 目录            卸载 weblogic 可执行命令及相关文件
|-/utils.jar 文件            工具 jar 文件
```

4.4　WebLogic home 目录

```
|-/common                    由产品组件共享的文件，包括用于设置在计算机
|                            上运行的所有 WebLogic 域所共用的环境属性、
|                            创建域时供 Configuration Wizard 和 WLST 脱机
|                            使用的模板 JAR 文件，以及来自第三方提供商的
|                            评估软件

|-/samples                   示例代码、资源和预配置的示例域，旨在帮助您
|                            学习如何使用产品软件开发您自己的应用程序。
|                            示例域按系统中安装的组件进行组织。例如，
|                            server 文件夹包含示例和 MedRec 示例应用程序
|                            的源代码

|-/server                    WebLogic Server 程序文件

|-/uninstall                 卸载 BEA Products 软件所需的代码
```

4.5　其他目录

WebLogic home 目录下的 samples 示例目录结构如下。

```
|samples
        |domains
              | medrec          medrec 应用程序示例域
              | wl_server       wl_server 应用程序示例域
        |server
              | docs            示例域的源代码
              | examples        随 weblogic 一起安装的示例
              |medrec           medrec 示例
```

第 5 章　WebLogic 配置相关文件

5.1　启动与服务相关的几个文件

WebLogic 的启动是通过启动文件来完成的，包括启动管理服务器（startWebLogic）、启动被管服务器（startManagedWebLogic）、设置域环境（setDomainEnv）、关闭管理服务器（stopWebLogic）、关闭被管服务器（stopManagedWebLogic）等。

5.1.1　setDomainEnv.cmd/setDomainEnv.sh

1．说明

setDomainEnv.cmd（Windows 环境下的启动文件），setDomainEnv.sh（Linux/UNIX 环境下的启动文件）。启动管理和被管服务器时的参数信息记录在此文件里。例如：设置堆内存的最小值和最大值-Xms256m、-Xmx512m，sun JDK 和 HP JDK 的 perm 区大小-XX:PermSize=48m，-XX:MaxPermSize=128m。

2．文件位置

/WLS_HOME/user_projects/domains/base_domain/bin。

5.1.2　startManagedWebLogic.cmd/startManagedWebLogic.sh

1．说明

startManagedWebLogic.cmd（Windows 环境下的启动文件），startManagedWebLogic.sh（Linux/UNIX 环境下的启动文件）。

2．文件位置

/WLS_HOME/user_projects/domains/base_domain/bin。

5.1.3　startWebLogic.cmd/startWebLogic.sh

1．说明

startWebLogic.cmd（Windows 环境下的启动文件），startWebLogic.sh（Linux/UNIX 环境下的启动文件）。

2. 文件位置

/WLS_HOME/user_projects/domains/base_domain（默认安装目录，可更改）启动文件的最外层目录，调用/WLS_HOME/user_projects/domains/base_domain/bin（默认安装目录，可更改）目录下的 startWebLogic.cmd/ startWebLogic.sh。

5.1.4 stopWebLogic.cmd/stopWebLogic.sh

1. 说明

stopWebLogic.cmd（Windows 环境下的启动文件），stopWebLogic.sh（Linux/UNIX 环境下的启动文件）。

2. 文件位置

/WLS_HOME/user_projects/domains/base_domain/bin。

5.1.5 stopManagedWebLogic.cmd/stopManagedWebLogic.sh

1. 说明

stopManagedWebLogic.cmd（Windows 环境下的启动文件），stopManagedWebLogic.sh（Linux/UNIX 环境下的启动文件）。

2. 文件位置

/WLS_HOME/user_projects/domains/base_domain/bin。

5.2 系统配置文件 config.xml

系统配置文件 config.xml 内存放着域内所有的配置信息，该文件的存放位置为/WLS_HOME/user_projects/domains/base_domain/config/config.xml（默认安装目录，其中域的名字以及路径可任意选择）。

5.2.1 WebLogic 管理和 config.xml 文件概述

WebLogic 的管理和配置服务基于 Sun 公司的 Java™ Management Extensions （JMX）API。创建 config.xml 文件的目的是存储管理对象的变更信息，以便确保在 WebLogic 重新启动时这些对象仍然可用。

通常应使用管理控制台来配置 WebLogic Server 的管理对象和服务以及允许 WebLogic

Server 来维持 config.xml 文件。每次通过管理控制台或其他 WebLogic Server 工具修改 config.xml 文件，WebLogic 都会把它归档到一个旧文件中，你可以配置 WebLogic 保存的归档文件的个数。

虽然 config.xml 是一个良好的 XML 文档，你可以使用文本编辑器修改，但你应该把它看作一个数据库，你只会在特殊情况下直接更新它。该文件并不是一个正式的 XML 内容文档，它实际上是一个库，其中每个 XML 元素包含了一个在 WebLogic Server 管理对象在内存中实例的请求保存的数据。

5.2.2　何时去编辑 config.xml

只有在很少的情况下您会选择直接修改文件而不是通过管理控制台。在这些情况下您要保证所的更改遵循最小化、明确化。您不能试图通过写一个新的 config.xml 文件来创建一个新的 WebLogic 配置。

警告：您不能在 WebLogic 运行期间修改 config.xml，因为 WebLogic 会定期重写 config.xml 文件。您的更改将会丢失并且根据您平台的不同可能会导致 WebLogic 服务的失败。在任何情况下您应该先保留一份 config.xml 的副本，再去修改文件。

这里有一些情况适合您直接修改 config.xml 文件。

- ❑ 如果您要部署多个 WebLogic Server 实例，您可以“克隆”一个 config.xml 文件并编辑新的服务器上的每个值。
- ❑ 如果您由一台服务器上定义一个对象，并希望将其复制到另一台服务器，您可以复制在 config.xml 文件中定义的 xml 元素到另一个 config.xml 文件中。
- ❑ 为了帮助您更好地解决问题，WebLogic 的技术支持可能会建议您在 config.xml 文件中设置一些不能通过管理控制台设定的属性。
- ❑ 一些第三方应用程序文件可能需要您修改 config.xml 文件。

5.2.3　config.xml 文件里的内容

config.xml 文件中包含了一系列的 xml 元素。其中域元素的顶层元素、域中所有元素都是域元素的子集。域元素包括服务器、集群、应用等子元素，这些元素可能还有其自己的子元素。例如，服务器的子元素包含 Web 服务器、SSL 和日志等，应用程序的子元素包含 EJB 组件和 Web 应用程序组件等。

每个元素都有一个或多个可配置的属性。一个属性有一个配置 API 中相应的属性。例如，服务器元素有一个 ListenPort 属性，同样，weblogic.management.configuration.ServerMBean 类有一个 ListenPort 属性。可配置的属性是可读可写的，也就是 ServerMBean 的 getListenPort()和 setListenPort()方法。

所有属性的值都必须加引号。Boolean 属性的值只能是 true 或 false。整数值不能包含逗号或小数点，但如果属性允许负值，可以有一个前置的减号。

5.3 属性文件 weblogic.xml

5.3.1 概要说明

WebLogic Server 允许通过设置 WebLogic 应用程序扩展描述符（weblogic.xml）配置 JSP 容器，该文件通常位于 Web 应用程序的 web-inf 目录下。可配置的元素包含在</weblogic-web-app></weblogic-web-app>根节点下。

5.3.2 可配置的属性详解

1. description

description 元素是 Web 应用程序的文字描述。

2. weblogic-version

weblogic-version 元素表示该 Web 应用程序（在根元素 <weblogic-web-app>下定义）即将部署至的 WebLogic Server 的版本。该元素仅表示版本信息，WebLogic Server 并不使用该元素。

3. security-role-assignment

security-role-assignment 元素声明 Web 应用程序安全角色与 WebLogic Server 中一个或多个委托人之间的映射，如以下示例所示。

示例 5-1：

```
<security-role-assignment>
    <role-name>PayrollAdmin</role-name>
    <principal-name>George</principal-name>
    <principal-name>simon</principal-name>
    <principal-name>system</principal-name>
</security-role-assignment>
```

还可以使用它将给定角色标记为外部定义的角色，如以下示例所示。

示例 5-2：

```
<security-role-assignment>
    <role-name>admin</role-name>
    <externally-defined/>
</security-role-assignment>
```

注 意

在 <security-role-assignment> 元素中，必须定义 <principal-name> 或 <externally-defined>，不能两者都忽略。

表 5-1 描述了在 security-role-assignment 元素中定义的元素。

表 5-1

元素	必需/可选	描述
<role-name>	必需	指定安全角色的名称
<principal-name>	如果未定义 <externally-defined>，则此元素必需	指定安全领域内定义的委托人的名称。可以使用多个 <principal-name> 元素向一个角色映射多个委托人
<externally-defined>	如果未定义 <principal-name>，则此元素必需	指定在安全领域内全局定义某特定安全角色；WebLogic Server 使用该安全角色作为委托人名称，而无需在全局领域内查找委托人。如果还在其他位置上定义了该安全角色及其委托人名称的映射，则该元素将用做表示性占位符

注 意

（1）如果您不定义 security-role-assignment 元素及其子元素，则 Web 应用程序容器会将该角色名隐式映射为委托人名称，并记录一条警告。如果不定义映射，则 EJB 容器不部署该模块。如果在 weblogic.xml 中将 role_xyz 映射至用户 joe，则 role_xyz 成为本地角色。 如果将 role_xyz 指定为外部定义的角色，则它将成为全局角色（它指领域级别上定义的角色）。

（2）如果不定义 security-role-assignment 元素，role_xyz 则成为本地角色，且 Web 应用程序容器创建其隐式映射，并记录一条警告。

4．run-as-role-assignment

run-as-role-assignment 元素将 web.xml 中的 run-as 角色名（servlet 元素的子元素）映射为系统中的有效用户名。对于给定 Servlet，该值将被的 Servlet-descriptor 中的 run-as-principal-name 元素替代。如果给定角色名中没有指定 run-as-role-assignment，则 Web 应用程序容器使用 security-role-assignment 中定义的第一个 principal-name。以下示例说明如何使用 run-as-role-assignment 元素。

示例 5-3：

```
<run-as-role-assignment>
<role-name>RunAsRoleName</role-name>
<run-as-principal-name>George</run-as-principal-name>
</run-as-role-assignment>
```

表 5-2 描述了可在 run-as-role-assignment 元素中定义的元素。

表 5-2

元素	必需/可选	描述
<role-name>	必需	指定安全角色的名称
<run-as-principal-name>	必需	指定委托人的名称

5．resource-description

resource-description 元素用于将服务器资源的 JNDI 名映射至 WebLogic Server 中的 EJB 资源引用。

表 5-3 描述了可在 resource-description 元素中定义的元素。

表 5-3

元素	必需/可选	描述
<res-ref-name>	必需	指定资源引用的名称
<jndi-name>	必需	指定资源的 JNDI 名

6．resource-env-description

resource-env-description 元素将 ejb-jar.xml 部署描述符中声明的 resource-env-ref 映射至其代表的服务器资源的 JNDI 名。

表 5-4 描述了可在 resource-env-description 元素中定义的元素。

表 5-4

元素	必需/必须	描述
<res-env-ref-name>	必需	指定资源环境引用的名称
<jndi-name>	必需	指定资源环境引用的 JNDI 名

7．ejb-reference-description

表 5-5 描述了可在 ejb-reference-description 元素中定义的元素。

表 5-5

元素	必需/必须	描述
<ejb-ref-name>	必需	指定 Web 应用程序中使用 EJB 引用的名称
<jndi-name>	必需	指定引用的 JNDI 名

8．service-reference-description

表 5-6 描述了可在 service-reference-description 元素中定义的元素。

表 5-6

元素	必需/可选	描述
<service-ref-name>		
<wsdl-url>		
<call-property>		<call-property> 元素具有下列子元素： （1）<name> （2）<value>
<port-info>		<port-info> 元素具有下列子元素： （1）<port-name> （2）<stub-property> （3）<call-property>

9．session-descriptor

session-descriptor 元素定义 Servlet 会话的参数（表 5-7）。

表 5-7

元素	默认值	值
<timeout-secs>	3600	设置 WebLogic Server 等待会话超时的时间（秒）。默认值为 3600 秒 在繁忙的站点上，可以通过调整会话超时时间来调整应用程序 Session 所占用的内存大小。尽管您希望为浏览器客户端提供每个完成会话的机会，但如果用户已离开站点或已放弃会话，您也不希望毫无必要地占用服务器器 该元素可以由 web.xml 中的 session-timeout 元素（分钟）替代
<invalidation-interval-secs>	60	设置 WebLogic Server 在执行超时会话和无效会话的清理检查与删除旧会话并释放内存之间需要等待的时间（秒）。使用此元素调整 WebLogic Server 以在高流量站点上获得最佳性能 默认值为 60 秒
<sharing-enabled>	false	如果该值在应用程序级别上设置为 true，则 Web 应用程序能够共享 HTTP 会话 如果在 Web 应用程序级别打开该元素，将忽略该元素
<debug-enabled>	false	启用 HTTP 会话的调试功能 默认值为 false
<id-length>	52	设置会话 ID 的大小 最小值为 8 字节，最大值为 Integer.MAX_VALUE 如果您正在编写 WAP 应用程序，您必须使用 URL 重写，因为 WAP 协议不支持 cookie。同时，某些 WAP 设备限制 URL 的长度（包括特性）不得超过 128 个字符，这限制了使用 URL 重写功能可以传输的数据量。为了给各个特性预留较多空间，可以使用此特性限制 WebLogic Server 随机生成的会话 ID 的大小 还可以通过设置 WAPEnabled 特性将长度限制为固定的 52 位字符，且不允许使用特殊字符 http://edocs.weblogicfans.net/wls/docs92/webapp/sessions.html - waphttp://edocs.weblogicfans.net/wls/docs92/webapp/sessions.html - wap
<tracking-enabled>	true	启用 HTTP 请求之间的会话跟踪
<cache-size>	1028	设置 JDBC 持久性会话和文件持久性会话的缓存大小

续表

元素	默认值	值
<max-in-memory-sessions>	–1	设置内存/复制会话的最大限制 如果不能对内存中 Servlet 会话的使用加以限制，那么，随着新会话的持续创建，服务器最终必然引发内存不足。为防止出现此问题，WebLogic Server 将针对所创建会话的数量提供可配置的限制。超出该数量时，每次尝试新建会话时都会引发 weblogic.servlet.SessionCreationException。此功能适用于复制内存中会话和非复制内存中会话 要对内存中 servlet 会话的使用加以限制，您可以在 max-in-memory-sessions 元素中设置限制 默认值为 –1（无限制）
<cookies-enabled>	true	默认情况下启用会话 cookie，建议保持此状态，但也可以通过将该属性设置为 false 来禁用它们。您可能会关闭该选项以进行测试
<cookie-name>	JSESSIONID	定义会话跟踪 cookie 名称如果没有设置，则默认值为 JSESSIONID。可以将其设置为适用于您的应用程序的更具体的名称
<cookie-path>	null	定义会话跟踪 cookie 路径 如果未设置此特性，则此特性默认为/（斜杠），浏览器会向 WebLogic Server 服务的所有 URL 发送 cookie。可以将该路径设置为限制性更强的映射，以便限制浏览器会向其发送 cookie 的请求 URL
<cookie-domain>	null	指定 cookie 有效的域。例如，将 cookie-domain 设置为.mydomain.com 会向*.mydomain.com 域中的所有服务器返回 cookie 域名必须至少由两部分组成，将名称设置为*.com 或*.net 是无效的 如果不设置此特性，则此特性默认为发出 cookie 的服务器 有关详细信息，请参阅 Sun Microsystems 的 Servlet 规范中的 Cookie.setDomain()
<cookie-comment>	null	指定标识 cookie 文件中会话跟踪 cookie 的注释
<cookie-secure>	false	指示浏览器仅通过 HTTPS 连接传回 cookie，这可以确保 cookie ID 是安全的，且仅用于使用 HTTPS 的网站。如果启用此功能，则 HTTP 上的会话 cookie 将不再起作用 如果希望使用此特性，则应禁用 url-rewriting-enabled 元素
<cookie-max-age-secs>	–1	设置客户端上的会话 cookie 的生命周期（秒），会话 cookie 超过该时间即会过期 默认值为 –1（无限制）

续表

元素	默认值	值
<persistent-store-type>	memory	将持久性存储方法设置为以下某个选项： （1）memory——禁用持久性会话存储 （2）replicated——与 memory 相同，但会话数据将在集群服务器之间复制 （3）replicated_if_clustered——如果 Web 应用程序部署于集群服务器上，则会复制生效的 persistent-store-type。否则，memory 为默认值 （4）sync-replication-across-cluster——复制将在集群内同步发生 （5）async-replication-across-cluster——复制将在集群内异步发生 （6）file——使用基于文件的持久性 （7）jdbc——使用数据库存储持久性会话 （8）cookie——所有会话数据都存储于用户浏览器的 cookie 中
<persistent-store-cookie-name>	WLCOOKIE	设置基于 cookie 的持久性所使用的 cookie 的名称。WLCOOKIE cookie 中带有会话状态，不应在 Web 应用程序之间共享
<persistent-store-dir>	session_db	指定基于文件的持久性所使用的存储目录，确保您有充足的磁盘存储空间（有效会话数与每个会话大小的乘积）。可以通过查看 persistent-store-dir 下创建的文件来确定会话的大小。注意：每个会话的大小随序列化会话数据的大小变化而变化 每个服务器实例都有一个默认的不需要配置的持久性文件存储。因此，如果不指定目录，会在 <server-name>\data\store\default 目录中自动创建默认存储。但是，默认存储不能在集群服务器间共享 可以在不同服务器之间共享的目录中创建自定义持久性存储，从而使文件持久性会话集群化。但是需要手工创建此目录
<persistent-store-pool>	None	指定要用于持久性存储的 JDBC 连接缓冲池的名称
<persistent-store-table>	wl_servlet_sessions	指定用于存储基于 JDBC 的持久性会话的数据库表名。仅当 persistent-store-type 设置为 JDBC 时，此项才适用 当您选择数据库表名称而非默认值时，将使用 persistent-store-table 元素
<jdbc-column-name-max-inactieinterval>		是 wl_max_inactive_interval 列名的备用名称。这个 jdbc-column-name-max-inactive-interval 元素仅适用于基于 JDBC 的持久性。某些不支持长列名的数据库需要使用该元素
<jdbc-connection-timeout-secs>	120	注意：本版本中不赞成使用该元素 设置 WebLogic Server 等待 JDBC 连接超时的时间（秒）

续表

元素	默认值	值
<url-rewriting-enabled>	true	启用 URL 重写（将会话 ID 编码到 URL 中），如果浏览器中禁用 cookie，它还可以提供会话跟踪
<http-proxy-caching-of-cookies>	true	设置为 false 时，WebLogic Server 会在下面的响应中添加如下头信息： "Cache-control: no-cache=set-cookie" 这表示代理缓存没有缓存 cookie
<encode-session-id-in-query-params>	false	最新 Servlet 规范要求容器将会话 ID 编码在路径参数中。某些 Web 服务器不太支持路径参数。在这样的情况下，应该将 encode-session-id-in-query-params 元素设置为 true。（默认值为 false。）
<runtime-main-attribute>		用于 ServletSessionRuntimeMBean 中。ServletSession-RuntimeMBean 的 getMainAttribute()会使用该字符串为关键字而返回会话特性值 示例：user-name 该元素适用于标记不同会话的会话运行时信息

10．jsp-descriptor

jsp-descriptor 元素指定 JSP 编译器的配置参数列表。表 5-8 描述了可在 jsp-descriptor 元素中定义的元素。

表 5-8

元素	必需/必须	描述
<page-check-seconds>	1	设置 WebLogic Server 检查 JSP 文件的时间间隔(秒)，以便查看 JSP 文件是否已更改以及是否需要重新编译。如果已发生更改，还会检查依赖关系并递归重新加载 （1）值–1 表示永不检查页面。该值为生产环境中的默认值 （2）值 0 表示总是检查页面 （3）值 1 表示每秒检查一次页面。该值为开发环境中的默认值 在生产环境下，很少发生 JSP 更改，可根据您的调整要求将 pageCheckSeconds 的值更改为 60 或更大
<precompile>	false	如果设置为 true，当部署或重新部署 Web 应用程序时，或启动 WebLogic Server 时，WebLogicServer 会自动预编译所有已修改的 JSP
<precompile-continue>	false	如果设置为 true，即使编译期间其中某些 JSP 失败，WebLogic Server 也会继续预编译所有已修改的 JSP。仅当 precompile 设置为 true 时才生效
<keepgenerated>	false	在 JSP 编译过程中保存作为中间步骤生成的 Java 文件除非此参数设置为 true，否则编译完成后将删除所有中间 Java 文件

续表

元素	必需/必须	描述
<verbose>	true	设置为 true 时，将向浏览器、命令提示和 WebLogic Server 日志文件中输出调试信息
<working-dir>	内部生成的目录	WebLogic Server 用于保存为 JSP 生成的 Java 和编译类文件的目录的名称
<print-nulls>	null	设置为 false 时，此参数确保表达式结果为 null 时将输出 “ ”
<backward-compatible>	true	设置为 true 时，支持向后兼容
<encoding>	平台的默认编码	指定 JSP 页面中使用的默认字符集 如果没有设置，则此特性默认为平台编码 JSP 页面指令（包含于 JSP 代码中）会替代此设置。例如，<%@ page contentType="text/html; charset=custom-encoding"%>
<package-prefix>	jsp_servlet	指定所有 JSP 页面将编译至的包的前缀
<exact-mapping>	true	设置为 true 时，只要第一次请求 JSP，新创建的 JspStub 即将准确地映射至该请求。如果 exactMapping 设置为 false，则 Web 应用程序容器将为 JSP 生成不准确的 URL 映射，exactMapping 允许提供 JSP 页面的路径信息
<default-file-name>	true	WebLogic Server 用于保存为 JSP 生成的 Java 和编译类文件的默认文件的名称
<rtexprvalue-jsp-param-name>	false	允许在 jsp:param 标记的 name 特性中使用运行时表达式值。它默认设置为 false

11. auth-filter

auth-filter 元素指定身份验证筛选器 HttpServlet 类。



当前版本中不赞成使用该元素，而改用 Servlet 身份验证筛选器。

12. charset-params

charset-params 元素用于定义非 unicode 操作的代码集行为。

示例 5-4：

```
<charset-params>
<input-charset>
<resource-path>/*</resource-path>
<java-charset-name>UTF-8</java-charset-name>
</input-charset>
</charset-params>
```

13．input-charset

使用 input-charset 元素定义用于读取 GET 和 POST 请求数据的字符集。

示例 5-5：

```
<input-charset>
  <resource-path>/foo</resource-path>
    <java-charset-name>SJIS</java-charset-name>
  </input-charset>
```

表 5-9 描述了可在<input-charset> 元素中定义的元素。

表 5-9

元素	必需/可选	描述
<resource-path>	必需	如果某请求的 URL 中包含此路径，则将指示 WebLogic Server 使用 java-charset-name 指定的 Java 字符集
<java-charset-name>	必需	指定要使用的 Java 字符集

14．charset-mapping

使用 charset-mapping 元素将 IANA 字符集名称映射至 Java 字符集名称。例如：

示例 5-6：

```
<charset-mapping>
    <iava-charset-name>Shift-JIS</iana-charset-name>
    <java-charset-name>SJIS</java-charset-name>
</charset-mapping>
```

表 5-10 描述了可在 charset-mapping 元素中定义的元素。

表 5-10

元素	必需/可选	描述
<iana-charset-name>	必需	指定 IANA 字符集名称，该名称将映射至 java-charset-name 元素指定的 Java 字符集中
<java-charset-name>	必需	指定要使用的 Java 字符集

15．virtual-directory-mapping

使用 virtual-directory-mapping 元素指定文档根，取代某些特定请求（例如图像请求）的 Web 应用程序的默认文档根。一组 Web 应用程序的所有图像可以存储于一个位置上，不需要复制到使用这些图像的每个 Web 应用程序的文档根中。对于传入的请求，如果已经指定虚拟目录，则 Servlet 容器将首先在该虚拟目录中搜索所请求的资源，然后再在 Web 应用程序的原始文档根下搜索。这定义了两个位置上具有同一文档时的搜索优先级。

示例 5-7：

```
<virtual-directory-mapping>
```

```
        <local-path>c:/usr/gifs</local-path>
        <url-pattern>/images/*</url-pattern>
        <url-pattern>*.jpg</url-pattern>
    </virtual-directory-mapping>
    <virtual-directory-mapping>
        <local-path>c:/usr/common_jsps.jar</local-path>
        <url-pattern>*.jsp</url-pattern>
    </virtual-directory-mapping>
```

表 5-11 描述了可在 virtual-directory-mapping 元素中定义的元素。

表 5-11

元素	必需/可选	描述
<local-path>	必需	指定磁盘上的物理位置
<url-pattern>	必需	包含映射的 URL 模式。必须遵循 Servlet API 规范的 11.2 部分中指定的规则

WebLogic Server 的虚拟目录映射实现要求您的目录包含映射的 URL 模式。图像示例要求您在 c:/usr/gifs/images 上创建名为 images 的目录。这允许 Servlet 容器在 images 目录下找到适用于多个 Web 应用程序的图像。

16．url-match-map

使用该元素指定用于 URL 模式匹配的类。WebLogic Server 默认 URL 匹配映射类是基于 J2EE 标准的 weblogic.servlet.utils.URLMatchMap。WebLogic Server 中包含的另外一个实现是 SimpleApacheURLMatchMap，可以通过使用 url-match-map 元素将该实现用做插件。

SimpleApacheURLMatchMap 的规则：

如果将*.jws 映射至 JWSServlet，则

http://foo.com/bar.jws/baz 将解析为路径信息为 baz 的 JWSServlet。

按照如下示例在 weblogic.xml 中配置要使用的 URLMatchMap。

示例 5-8：

```
<url-match-map>
    weblogic.servlet.utils.SimpleApacheURLMatchMap
</url-match-map>
```

17．security-permission

security-permission 元素根据安全策略文件语法指定单个安全权限。

放弃可选的 codebase 和 signedBy 子句。

示例 5-9：

```
<security-permission-spec>
    grant { permission java.net.SocketPermission "*", "resolve" };
</security-permission-spec>
```

其中：

permission java.net.SocketPermission 是权限类名。

"*" 表示目标名称。

resolve 表示操作。

18. context-root

context-root 元素定义该独立 Web 应用程序的根路径。如果 Web 应用程序不是独立的，而属于某 EAR 的一部分，则应在该 EAR 的 META-INF/application.xml 文件中指定 context-root。application.xml 中的 context-root 设置优先于 weblogic.xml 中的 context-root 设置。



> 该 weblogic.xml 元素仅用于使用两阶段部署模型进行部署时。

确定 Web 应用程序的根路径的优先级顺序如下。

（1）在 application.xml 中检查 context-root，如果找到，则将其用做 Web 应用程序的根路径。

（2）如果 application.xml 中未设置 context-root，且 Web 应用程序是作为 EAR 的一部分部署的，则检查 weblogic.xml 中是否定义了 context-root。如果找到，则将其用做 Web 应用程序的根路径。如果 Web 应用程序是独立部署的，则 application.xml 不参与该过程，而仅从 weblogic.xml 开始 context-root 的确定，如果其中没有定义，则默认使用 URI。

（3）如果 weblogic.xml 或 application.xml 中没有 context-root，则将从 URI 中推断上下文路径，将 URI 中定义的值减去 WAR 后缀作为其名称。例如，如果 URI 为 MyWebApp.war，则其根路径为/MyWebApp。

注 意

> 不能为 EAR 库中的各个 Web 应用程序分别设置 context-root 元素。只能针对 Web 应用程序库进行设置。

19. wl-dispatch-policy

使用 wl-dispatch-policy 元素，通过标识执行队列名称，从而将 Web 应用程序分配至已配置的执行队列。可以使用 per-servlet-dispatch-policy 元素在单个 Servlet 或 JSP 级别上替代该 Web 应用程序级别参数。

20. servlet-descriptor

使用 servlet-descriptor 元素聚合 servlet 特定的元素。

表 5-12 描述了可在 servlet-descriptor 元素中定义的元素。

表 5-12

元素	必需/可选	描述
<servlet-name>	必需	将 servlet 名称指定为 web.xml 部署描述符文件的 servlet 元素中定义的 servlet 名称
<run-as-principal-name>	可选	包含针对 web.xml 部署描述符中所定义的 run-as-role-name 的委托人名称
<init-as-principal-name>	可选	相当于 servlet 的 init 方法的 run-as-principal-name。此处指定的标识应为系统内的有效用户名。如果未指定 init-as-principal-name，则容器将使用 run-as-principal-name 元素
<destroy-as-principal-name>	可选	相当于 servlet 的 destroy 方法的 run-as-principal-name。此处指定的标识应为系统内的有效用户名。如果未指定 destroy-as-principal-name，则容器将使用 run-as-principal-name 元素
<dispatch-policy>	可选	不赞成使用此元素。用于通过标识执行队列名称来向某个已配置的执行队列分配给定的 servlet。该设置将替代 wl-dispatch-policy 定义的 Web 应用程序级别调度策略

21. work-manager

work-manager 元素是<weblogic-web-app>元素的子元素。可以在 work-manager 元素中定义表 5-13 所列的元素。

表 5-13

元素	必需/可选	描述
<name>	必需	指定工作管理器的名称
<response-time-request-class/ fair-share-request-class/ context-request-class/ request-class-name>	可选	可以从以下 4 个元素之间选择一个： （1）response-time-request-class——定义应用程序的响应时间请求类。响应时间（毫秒）由特性 goal-ms 定义。增量为((目标˜ T) C_r)/R，其中 T 指平均线程使用时间，R 指达到率，C_r 指确定响应时间目标优先于公平共享的系数 （2）fair-share-request-class——定义公平共享请求类。公平共享由默认共享的属性百分比定义。因此，默认值为 100。增量为 C_f/(P R T)，其中 P 指百分比，R 指达到率，T 指平均线程使用时间，C_f 指公平共享优先级低于响应时间目标的系数 （3）context-request-class——定义上下文类。上下文由将上下文信息（如当前用户或其角色、cookie 或工作区域字段）映射到已命名的服务类的多个案例定义 （4）request-class-name——定义请求类名称
<min-threads-constraint / min-threads-constraint-name>	可选	可以从以下两个元素之间选择一个： （1）min-threads-constraint——用来保证服务器向受约束工作集的请求分配的线程数，以避免死锁。默认值为零。例如，对于复制更新请求，最少线程值如果唯一（可以从对等方同步调用），则该值非常有用 （2）min-threads-constraint-name——定义 min-threads-constraint 元素的名称

续表

元素	必需/可选	描述
<max-threads-constraint / max-threads-constraint-name>	可选	可以从以下两个元素之间选择一个： （1）max-threads-constraint——限制执行来自受约束工作集的请求的并发线程数量。默认值为无限制。例如，假设约束被定义为最多具有 10 个线程，并且由 3 个入口点共享。调度逻辑可确保执行来自 3 个入口点组合的请求的线程不超过 10 个 （2）max-threads-constraint-name——定义 max-threads-constraint 元素的名称
<capacity / capacity-name>	可选	可以从以下两个元素之间选择一个： （1）capacity——可以定义约束并将其应用于入口点集（称为受约束的工作集）。只有达到该容量时服务器才开始拒绝请求，默认值为零。注意：容量包括受约束的工作集中已经排队或正在执行的所有请求。此约束主要用于像 JMS 之类的子系统，这些子系统执行自己的流控制。此约束与全局队列阈值无关 （2）capacity-name——定义 capacity 元素的名称

22. logging

logging 元素是 <weblogic-web-app> 元素的子元素。可以在 logging 元素中定义表 5-14 所列的元素。

表 5-14

元素	必需/可选	描述
<log-filename>	必需	指定日志文件的名称。需要提供该文件名的完整地址
<logging-enabled>	可选	指定是否为 ManagedConnectionFactory 或 ManagedConnection 设置了日志编写器。如果将此元素设置为 true，则从 ManagedConnectionFactory 或 ManagedConnection 中生成的输出将发送到由 log-filename 元素指定的文件。如果不指定此值，则 WebLogic Server 将使用其定义的默认值。值范围：true | false 默认值：false
<rotation-type>	可选	设置文件滚动类型 值为 bySize、byName 或 none （1）bySize——当日志文件大小达到在 file-size-limit 中指定的大小时，服务器将把该文件重命名为 FileName.n （2）byName——按照在 file-time-span 中指定的时间间隔，服务器将该文件重命名为 FileName.n。在服务器重命名某个文件之后，后续的消息会累积在一个名称为 log-filename 中指定的新文件中 （3）none——消息累积在单个文件中。当文件体积过大时，必须清除它的内容 默认值：bySize

续表

元素	必需/可选	描述
<number-of-files-limited>	可选	指定是否应该对该服务器实例为存储旧消息而创建的文件数进行限制（需要指定 bySize 的滚动类型）。服务器达到此限制后，它将覆盖最旧的文件。如果不启用此选项，服务器将无限制地创建新文件，这样就必须在需要时清除这些文件 如果通过将 number-of-files-limited 设置为 true 启用该选项，则服务器将参考 rotationType 变量来决定如何滚动该日志文件。滚动意味着替换现有文件而不是创建新文件。如果将 number-of-files-limited 指定为 false，则服务器将创建大量日志文件而不是替换同一个日志文件 值范围：true | false 默认值：false
<file-count>	可选	服务器在滚动日志时创建的日志文件的最大数量。该数目不包括服务器用于存储当前消息的文件（需要启用 number-of-files-limited） 默认值：7
<file-size-limit>	可选	触发服务器将日志消息移动到单个文件的大小。（需要指定 bySize 的滚动类型。）日志文件达到指定的大小之后，服务器会在下次检查文件大小时将当前的日志文件重命名为 FileName.n，并新建一个用来存储后续消息的文件 默认值：500
<rotate-log-on-startup>	可选	指定服务器在其启动周期内是否滚动其日志文件 值范围：true | false 默认值：true
<log-file-rotation-dir>	可选	指定将存储滚动日志文件的目录的路径
<rotation-time>	可选	日志文件基于时间滚动顺序的开始时间的格式为 k:mm，其中 k 为 1～24。（需要指定 byTime 的滚动类型。）在到达指定时间时，服务器会重命名当前的日志文件。之后，服务器将按照在 file-time-span 中指定的间隔重命名日志文件 如果指定的时刻已过，服务器会立即开始其文件滚动 默认情况下，滚动周期会立即开始
<file-time-span>	可选	服务器将旧的日志消息保存到另一个文件的时间间隔（小时）（需要指定 byTime 的滚动类型） 默认值：24

23. library-ref

library-ref 元素引用计划用做当前 Web 应用程序中的 Web 应用程序库的一个库模块。示例 5-10：

```
<library-ref>
    <library-name>WebAppLibraryFoo</library-name>
    <specification-version>2.0</specification-version>
```

```
        <implementation-version>8.1beta</implementation-version>
        <exact-match>false</exact-match>
    </library-ref>
```

只有下列子元素与 Web 应用程序相关：library-name、specification-version、implementation-version 以及 exact-match。

可以在 library-ref 元素中定义表 5-15 所列的元素。

表 5-15

元素	必需/可选	描述
<library-name>	必需	提供用于库模块引用的库名称。默认值为 null
<specification-version>	必需	提供用于库模块引用的规范版本。默认值为 0（为浮点型）
<implementation-version>	必需	提供用于库模块引用的实现版本。默认值为 null
<exact-match>	必需	默认值为 false

24. Backwards Compatibility Flags

本版本中增添了若干向后兼容性标志，允许您恢复 WebLogic Server 9.0 之前的版本中所见的行为。

25. Web Container Global Configuration

要在全局级别上配置 Web 容器，可以使用 WebAppContainerMBean。

5.4 属性文件 web.xml

5.4.1 概要说明

Web 工程中，使用 web.xml 文件来配置欢迎页面、servlet、filter 等 Web 元素。web.xml 的模式文件是由 JavaEE 对应的规范定义的，每个 web.xml 文件的根元素在<web-app>中都必须标明这个 web.xml 使用的是哪个模式文件。

示例 5-11：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    Xmlns=http://jaava.sun.com/xml/ns/javaee
    Xmlns:xsl=http://www.w3.org/2001/XMLSchema-instance
    Xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
</web-app>
```

而且 web.xml 的模式文件中定义的标签并不是死的，模式文件也是可以改变的，一般

来说，随着 web.xml 模式文件的版本升级，里面定义的功能会越来越复杂，也就是标签元素的种类会越来越多，但有些是不常用的，我们只需记住一些常用的就可以了。

5.4.2　可配置的属性详解

1．icon

可以给应用指定一大一小两个图标（表 5-16）。

表 5-16

元素	必须/可选	描述
<small-icon>	可选	小图标的位置建议采用 16×16 像素的 gif 或 jpg 文件
<large-icon>	可选	小图标的位置建议采用 32×32 像素的 gif 或 jpg 文件

2．display-name

指定 Web 应用程序的显示名称（表 5-17）。

表 5-17

元素	必须/可选	描述
<display-name>	可选	

3．description

提供了有关 Web 应用程序的一些说明性文字（表 5-18）。

表 5-18

元素	必须/可选	描述
<description>	可选	

4．context-param

声明一个 Web 应用程序的 servlet 上下文初始化参数。表 5-19 描述了 WebLogic Server 保留的一些上下文参数（这些参数已经过时，并在 weblogic.xml 中有其他元素来替代）。

表 5-19

元素	必须/可选	描述
<weblogic.httpd.clientCertProxy>	可选	此属性指定 WebLogic 代理插件或者 HttpClusterServlet 会将 HTTPS 连接中客户端提供的证书在名称为 WL-Proxy-Client-Cert 的 HTTP 头中上传到服务器上

5．filter

定义了一个过滤器（Servlet Filter）和它的初始化属性。表 5-20 列出了 filter 的子元素。

表 5-20

元素	必须/可选	描述
<icon>	可选	图标
<filter-name>	必须	指定过滤器的名称，部署描述符的其他部分将使用过滤器的名字来引用这个过滤器
<display-name>	可选	显示名称
<description>	可选	过滤器的文字说明
<filter-class>	必须	过滤器的完整类名，包含包名
<init-param>	可选	包含一些键/值对作为过滤器的初始化参数

6．filter-mapping

指定过滤器生效的 URL（表 5-21）。

表 5-21

元素	必须/可选	描述
<filter-name>	必须	过滤器的名称，和<filter>元素里的<filter-name>对应
<url-pattern>	必须	指定过滤器所影响的 URL

7．listener

定义一个应用程序监听器（表 5-22）。

表 5-22

元素	必须/可选	描述
<listener-class>	可选	Web 应用事件监听器的完整类名

8．servlet

声明一个 HTTP Servlet，表 5-23 中列出了其子元素。

表 5-23

元素	必须/可选	描述
<icon>	可选	图标
<servlet-name>	必须	指定 Servlet 的名称；部署描述符的其他部分会采用这个名称来引用这个 Servlet
<display-name>	可选	显示名称
<description>	可选	Servlet 的文字说明
<servlet-class>	必须（或使用<jsp- file>）	Servlet 的完整类名。一个 Servlet 内，只能使用一个<servlet-class>标签或<jsp-file>标签
<jsp-file>	必须（或使用<servlet- class>）	JSP 文件相对于 Web 应用程序根目录的完整的路径
<init-param>	可选	包含一些键/值对作为 Servlet 的初始化参数

续表

元素	必须/可选	描述
<load-on-startup>	可选	指定当 WebLogic Server 启动时，是否初始化该 Servlet。其内容必须是正整数，指示该 Servlet 的加载顺序，越小的值越先被加载
<run-as>	可选	指定该 Servlet 运行时的身份，包含一个可选的描述和一个安全角色的名称
<security-role-ref>	可选	用于建立应用程序中使用的硬编码的安全角色名和部署描述符里<security-role>标签所定义的安全角色的映射

9．servlet-mapping

指定 Servlet 所处理的 URL，表 5-24 列出了其子元素。

表 5-24

元素	必须/可选	描述
<servlet-name>	必须	Servlet 的名字，对应<servlet>元素中声明的<servlet-name>
<url-pattern>	必须	Servlet 处理的 URL 模式

10．session-config

指定 Web 应用和 HTTP Session 相关的配置，表 5-25 描述了其子元素。

表 5-25

元素	必须/可选	描述
<session-timeout>	可选	指定 Web 应用程序中 HTTP 会话过期时间，单位是分钟。此处设置的值将覆盖 weblogic.xml 中<session-descriptor>的 TimeoutSecs 属性 默认值：–2 最大值：Integer.MAX_VALUE/60 特殊值： （1）–2=使用 weblogic.xml 中<session-descriptor>的 TimeoutSecs 属性值 （2）–1=会话不超时

11．mime-mapping

定义文件扩展名和 mime type 之间的映射，表 5-26 描述了其子元素。

表 5-26

元素	必须/可选	描述
<extension>	必须	文件扩展名，比如 txt
<mime-type>	必须	MIME 类型，比如 text/plain

12．welcome-file-list

指定欢迎页面的列表；当客户端请求的 URL 是一个目录名时，WebLogic 服务器根据这个列表来返回一个缺省页面（表 5-27）。

表 5-27

元素	必须/可选	描述
<welcome-file>	可选	默认欢迎页面的文件名，例如 index.html

13．error-page

错误页配置；当错误发生时，服务器可以根据异常或者 HTTP 响应的状态码，将对应的页面返回给客户端（表 5-28）。



可以使用<error-code>或<exception-type> 之一，但二者不能同时使用。

表 5-28

元素	必须/可选	描述
<error-code>	可选	有效的 HTTP 错误代码，例如 404
<exception-type>	可选	一个 Java 异常类的完整类名，例如 java.lang.Exception
<location>	必须	错误页面资源的位置，例如/myErrorPg.html

14．taglib

定义一个 JSP 标签库（表 5-29）。

表 5-29

元素	必须/可选	描述
<taglib-location>	必须	标记库描述符文件相对于 Web 应用程序根目录的文件路径。建议放置在 WEB-INF 目录下
<taglib-uri>	必须	指定 JSP 标签库的前缀 URI

15．resource-env-refresource-env-ref

定义一个对应用服务器环境资源对象的引用。

示例 5-12：

```
<resource-env-ref>
    <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

其中的元素及描述见表 5-30。

表 5-30

元素	必须/可选	描述
<description>	可选	描述
<resource-env-ref-name>	必须	指定该资源的引用名称，在应用程序中可以使用 java:comp/env 作为根路径，使用 JNDI 来使用该资源
<resource-env-ref-type>	必须	指定资源的类型，需要是类或接口的全名

16．resource-ref

定义了一个到外部资源的引用。这使得在 Web 应用代码里边可以使用一个“虚拟”的 JNDI 名字来引用一个外部资源，而其具体的资源可以在 weblogic.xml 里指定（表 5-31）。

表 5-31

元素	必须/可选	描述
<description>	可选	文字说明
<res-ref-name>	必须	Web 应用程序中使用这个 JNDI 名称来查找一个该资源
<res-type>	必须	该资源引用名称对应的 Java 类型，使用 Java 类型的完整类名
<res-auth>	必须	
<res-sharing-scope>	可选	Specifies whether connections obtained through the given resource manager connection factory reference can be shared.指定是否通过给定的资源管理器连接工厂引用获得连接可以共享 有效值： （1）Shareable 可共享 （2）Unshareable 不可共享

17．security-constraint

指定对<web-resource-collection>元素中定义的 Web 资源集合的访问权限（表 5-32）。

表 5-32

元素	必须/可选	描述
<web-resource-collection>	必须	指定该安全约束所应用的 HTTP 请求范围
<auth-constraint>	可选	指定哪些用户/组/角色对该 Web 资源的集合有访问权限
<user-data- constraint>	可选	指定客户端应该如何与服务器通信

18．web-resource-collection

定义的 Web 应用中安全约束的应用范围（表 5-33）。

表 5-33

元素	必须/可选	描述
<web-resource-name>	必须	Web 资源集合的名称
<description>	可选	文字描述
<url-pattern>	可选	
<http-method>	可选	GET/POST

19．auth-constraint

定义哪些组或角色获得此安全约束集合中定义的 Web 资源（表 5-34）。

表 5-34

元素	必须/可选	描述
<description>	可选	文字描述

20．user-data-constraint

定义客户端应该如何与服务器通信（表 5-35）。

表 5-35

元素	必须/可选	描述
<description>	可选	文字说明
<transport- guarantee>	必须	指定客户端和服务器之间的通信安全强度。当指定 INTEGRAL 或 CONFIDENTIAL 时，WebLogic 服务器要求客户端必须使用 SSL/HTTPS 来访问指定的资源集合 值范围： （1）NONE （2）INTEGRAL （3）CONFIDENTIAL

21．login-config

指定用户登录相关的选项，设置了该项后，当用户访问通过<security-constraint>设置的受限资源时，必须先通过用户认证，认证后，才可以访问被授权访问资源（表 5-36）。

表 5-36

元素	必须/可选	描述
<auth-method>	可选	指定用于验证用户身份的方法，可选的值： （1）BASIC——默认值，采用浏览器的 HTTP Basic 认证方式 （2）FORM——用户提交 HTML 表单方式进行认证 （3）CLIENT-CERT——客户端 HTTPS 证书认证
<realm-name>	可选	验证使用的安全域
<form-login- config>	可选	<auth-method>为 FORM 时使用

22．form-login-config

FORM 表单登录配置（表 5-37）。

表 5-37

元素	必须/可选	描述
<form-login-page>	必须	用于登录验证的资源 URI，可以是一个 HTML 页面、JSP 或 HTTP Servlet，必须返回一个 HTML 页面，其中包含一个符合特定的命名约定的身份验证表单
<form-error-page>	必须	身份验证失败时返回客户端的页面 URI

23．security-role

安全角色定义（表 5-38）。

表 5-38

元素	必须/可选	描述
<description>	可选	描述
<role-name>	必须	角色的名称。必须在 weblogic.xml 里有对应的条目

24．env-entry

环境参数设置（表 5-39）。

表 5-39

元素	必须/可选	描述
<description>	可选	描述
<env-entry-name>	必须	环境参数的名称
<env-entry-value>	必须	环境参数的值
<env-entry-type>	必须	环境参数的类型，可设置为以下 Java 类型之一： java.lang.Boolean java.lang.String java.lang.Integer java.lang.Double java.lang.Float

25．ejb-ref

EJB 映射的定义（表 5-40）。

表 5-40

元素	必须/可选	描述
<description>	可选	描述
<ejb-ref-name>	必须	Web 应用程序中引用的 EJB 的 JNDI 名称
<ejb-ref-type>	必须	预期的 EJB 的类型
<home>	必须	EJB 的 HOME 接口的全名
<remote>	必须	EJB 的远程接口的全名
<ejb-link>	可选	J2EE 应用程序包（ear）中包含一个 EJB 的名称
<run-as>	可选	必须是<security-role>元素所定义的一个安全角色

26．ejb-local-ref

定义一个对 EJB Local Home 接口的引用（表 5-41）。

表 5-41

元素	必须/可选	描述
<description>	可选	描述
<ejb-ref-name>	必须	Web 应用程序中引用的 EJB 的 JNDI 名称
<ejb-ref-type>	必须	预期的 EJB 的类型。必须是下列之一： <ejb-ref-type>Entity</ejb-ref-type> <ejb-ref-type>Session</ejb-ref-type>
<local-home>	必须	EJB 的本地 Home 接口的全名
<local>	必须	EJB 的本地接口的全名
<ejb-link>	必须	J2EE 应用程序包（ear）中包含一个 EJB 的名称

5.5 日志文件

5.5.1 域日志

存放位置：/WLS_HOME/userprojects/domains/base_domain/servers/AdminServer/logs（默认位置）

域日志记录一个 Domian 的运行情况，一个 Domain 中的各个 WebLogic Server 可以把它们的一些运行信息（比如很严重的错误）发送给一个 Domain 的 Administrator Server 上，Administrator Server 再把这些信息写到 Domain 日志中。默认名为 domain_< name >.log

5.5.2 服务器日志 server.log

存放位置：/WLS_HOME/user_projects/domains/base_domain/servers/AdminServer/logs/ WLS_HOME/user_projects/domains/base_domain/servers/servername/logs（默认位置）

WebLogic Server 运行日志。假如 WebLogic Server 在启动或运行过程中有错误发生，错误信息会显示在屏幕上，并且会记录在一个 LOG 文件中，该文件默认名为 AdminServer.log。该文件也记录 WebLogic 的启动及关闭等其他运行信息。可在 Gernal 属性页中设置该文件的路径及名字、错误的输出的等级等。同时创建的每个被管服务器也会有一个 server_< name >.log 文件记录被管服务器在运行时的信息。

5.5.3 访问日志 access.log

存放位置：/WLS_HOME/user_projects/domains/base_domain/servers/servername/logs（默认位置）

HTTP 访问日志。在 WebLogic 中可以对用 HTTP、HTTPS 协议访问的服务器上的文件都做记录，该 LOG 文件默认的名字为 Access.log，内容如下，该文件具体记录在某个时间、某个 IP 地址的客户端访问了服务器上的哪个文件。

127.0.0.1 - - [25/Feb/2002:11:35:58 +0800]"GET /landingbj HTTP/1.1" 302 0

127.0.0.1 - - [25/Feb/2002:11:35:58 +0800]"GET/landingbj/index.Html HTTP/1.1" 200 176
HTTP 访问日志的属性可在 HTTP 属性页中进行设置。

第 6 章　Java 虚拟机（JVM）相关知识

6.1　JVM 简介

Java Virtual Machine（Java 虚拟机），它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。Java 虚拟机有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。JVM 屏蔽了与具体操作系统平台相关的信息，使得 Java 程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行。

6.2　常见 JDK 的内存机制

Java 的内存由 3 个代组成，如图 6-1 所示。这 3 个代分别如下。

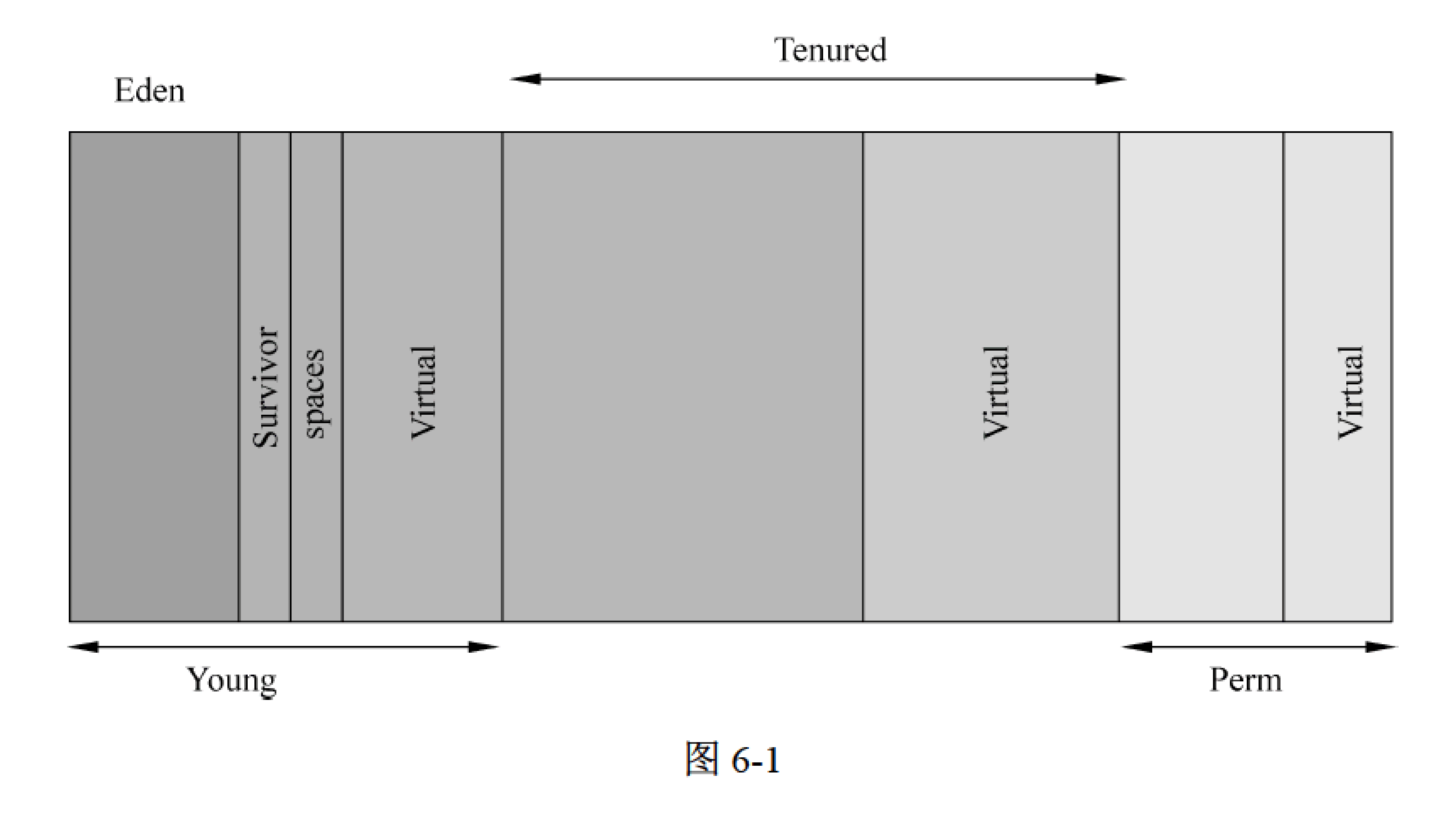


图 6-1

1．Young（年青代，也叫 new 区）

Young 还可以分为 Eden 区和两个 Survivor 区（from 和 to，这两个 Survivor 区大小严格一致），新的对象实例总是首先放在 Eden 区，Survivor 区作为 Eden 区和 Tenure（old）的缓冲，可以向 Tenure（old）转移活动的对象实例。

2．Tenured（保持代，也叫 old 区）

Tenure 中存放生命周期长久的实例对象，但是里面的对象也是会被回收掉的。

Young 和 Tenured 组成了 Java 的堆内存（Heap）。

3．Perm（永久代）

Perm（perm）则是非堆内存的组成部分。主要存放加载的 Class 类级对象如 class 本身，method、field 等。

在 JVM 启动时，就已经保留了固定的内存空间给 Heap 内存，这部分内存并不一定都会被 JVM 使用，但是可以确定的是这部分保留的内存不会被其他进程使用。这部分内存大小由-Xmx 参数指定。

影响这个的参数是-Xms，如果-Xms 指定的值比-Xmx 的小，那么两者的差值就是 Virtual 内存值。随着程序的运行，Eden 区、Tenured 区和 Perm 区会逐渐使用保留的 Virtual 空间。

6.3　Java 的 GC（Garbage Collection）原理

GC（Garbage Collection）是垃圾回收机制，在 Java 中开发人员无法使用指针来自由地管理内存，GC 是 JVM 对内存（实际上就是对象）进行管理的方式。GC 使得 Java 开发人员摆脱了烦琐的内存管理工作，让程序的开发更有效率。关于 GC 更详细的信息，可参考后续“虚拟机 GC 及其相关问题”章节。

6.4　JVM 中的 ClassLoader

6.4.1　ClassLoader 概述

类加载器（ClassLoader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。每个这样的实例用来表示一个 Java 类。

与 C 或 C++编写的程序不同，Java 程序是由许多独立的类文件组成的，每个文件对应于一个 Java 类。此外，这些类文件并非立即全部装入内存，而是根据程序需要装入内存的。ClassLoader 便是 JVM 中将类装入内存的那个零件。用户可以定制自己的 ClassLoader。JVM 中缺省的 ClassLoader 可以完成将本地文件系统装入类文件的任务。用户定制的 ClassLoader 可以拥有 JVM 缺省的 ClassLoader 所不具有的功能。例如，用户可以使用自己创建的 ClassLoader 从非本地硬盘或者从网络装入可执行内容。

JVM 在运行时会产生 3 个 ClassLoader，即 Bootstrap ClassLoader、Extension ClassLoader、App ClassLoader。BootStrap ClassLoader 使用 C++编写，屏幕上打印它时为 null。它用来在 JVM 启动时自动加载核心类库，运行过程中不能修改 Bootstrap 加载路径。Extension ClassLoader 用来加载 JRE\lib\ext 目录下的库文件，JRE\classes 目录下的类。App ClassLoader 用来加载 CLASSPATH 变量定制路径下的类。三者的层次关系如下。

示例 6-1：

```
BootStrap ClassLoader
|------ Extension ClassLoader
          |------- App ClassLoader
```

ClassLoader 加载类用的是委托模型，加载一个类时，首先 BootStrap 先进行寻找，找不到再由 ExtClassLoader 寻找，最后才是 AppClassLoader。

6.4.2 Java 类加载器

类加载器是 Java 语言的基本模块。类加载器是 Java 虚拟机的一部分，它会将类加载到内存中；类加载器负责在运行时查找和加载类文件。每个成功的 Java 编程人员都需要了解类加载器及其行为。本部分概述 Java 类加载器。

1. Java 类加载器层次结构

类加载器包含具有父类加载器和子类加载器的层次结构。父类加载器和子类加载器之间的关系类似于超类和子类之间的对象关系。下面是 3 种 JVM 提供的类加载器。

- **引导类加载器（bootstrap class loader）** 它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自 java.lang.ClassLoader。
- **扩展类加载器（extensions class loader）** 它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- **系统类加载器（system class loader）** 它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

除了引导类加载器之外，所有的类加载器都有一个父类加载器。通过 getParent()方法可以得到。对于系统提供的类加载器来说，系统类加载器的父类加载器是扩展类加载器，而扩展类加载器的父类加载器是引导类加载器；对于开发人员编写的类加载器来说，其父类加载器是加载 Java 类的类加载器。因为类加载器 Java 类如同其他的 Java 类一样，也是要由类加载器来加载的。一般来说，开发人员编写的类加载器的父类加载器是系统类加载器。

> **注 意**
>
> 在 WebLogic Server 之外的上下文中，Oracle 所指的“系统类路径类加载器”通常被称做“应用程序类加载器”。在 WebLogic Server 中讨论类加载器时，Oracle 使用术语“系统”，从而与 J2EE 应用程序或库相关的类加载器（Oracle 将其称做“应用程序类加载器”）区分开来。

2. 加载类

类加载器加载类时由于用到了代理模式，类加载器会首先代理给其他类加载器来尝试加载某个类。这就意味着真正完成类的加载工作的类加载器和启动这个加载过程的类加载器有可能不是同一个。真正完成类的加载工作是通过调用 defineClass 来实现的，而启动类

的加载过程是通过调用 loadClass 来实现的。前者称为一个类的定义加载器（defining loader），后者称为初始加载器（initiating loader）。在 Java 虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。也就是说，哪个类加载器启动类的加载过程并不重要，重要的是最终定义这个类的加载器。两种类加载器的关联之处在于：一个类的定义加载器是它引用的其他类的初始加载器。如类 com.example.Oute 引用了类 com.example.Inner，则由类 com.example.Outer 的定义加载器负责启动类 com.example.Inner 的加载过程。

类加载器在成功加载某个类之后，会把得到的 java.lang.Class 类的实例缓存起来。下次再请求加载该类的时候，类加载器会直接使用缓存的类的实例，而不会尝试再次加载。也就是说，对于一个类加载器实例来说，相同全名的类只加载一次，即 loadClass 方法不会被重复调用。

在 WebLogic Server 中可以对与 Web 应用程序关联的类加载器进行配置，使其首先在本地进行检查，然后再要求其父类加载器提供该类。这样，Web 应用程序能够使用其自己版本的第三方类，即便这些类也可能包含于 WebLogic Server 产品中。

3．更改正在运行的程序中的类

通过 WebLogic Server，您可以在服务器运行期间部署更新版本的应用程序模块，例如 EJB。这个过程即是所谓的热部署或热重新部署，它与类加载紧密相关。

Java 类加载器没有用于取消部署或卸载一组类的标准机制，也不能加载新版本的类。为更新正在运行的虚拟机中的类，必须将已加载该更改类的类加载器替换为新的类加载器。替换类加载器时，必须重新加载通过该类加载器（或该类加载器的任何子类加载器）加载的所有类。这些类的所有实例都必须重新实例化。

在 WebLogic Server 中，每个应用程序都具有类加载器的层次结构，它们是系统类加载器的子类加载器。这些层次结构允许单独重新加载应用程序或部分应用程序，而不会影响系统的其他部分。WebLogic Server 应用程序类加载中将讨论此主题。

6.4.3　WebLogic 类加载器

WebLogic 中发布一个应用一般的目录是：

```
myapplication
|---APP-INF            //放在这个目录下的 lib 和 classes 不能实例化 webapp
     |---lib           //放 ejb 和 webapp 公用的 jar 包
    |---classes        //放 ejb 和 webapp 公用的类
|---META-INF
    |---application.xml
|---mywebapp
     |---WEB-INF
    |---lib
    |---classes        //放 class 类
     |---web.xml
|---ejb.jar            //ejb 的 jar 包
```

对应的各层级 ClassLoader 如下：

```
     BootStrap ClassLoader
|--- Extension ClassLoader
             |---Weblogic Service System ClassLoader
                 |---Filtering ClassLoader
                    |---Application ClassLoader
                       |---Web Application ClassLoader
                          |---Jsp ClassLoader
```

Application ClassLoader 用来加载 EJB JARS、APP-INF/lib、APP-INF/classes、EJB JARS 中 ClassPath 变量定制的路径下的类。Web Application ClassLoader 用来加载 WAR、WAR 中 ClassPath 变量定制的路径下的类。WebLogic 下 ClassLoader 用的也是委托模型，首先 BootStrap 先进行寻找，找不到再由 ExtClassLoader 寻找，然后再由 AppClassLoader 一级一级往下找。这样的分层结构有一个好处，就是在 JSP、Servlet 中可以直接访问 EJB 的接口。这种上层装载 EJB，下层装载 servlet 等，最下面装载 jsp 文件的结构，使得经常变动的 JSP、Servlet 等可以被重新装载而不会涉及到 EJB 层。在 WebLogic 中可以通过修改配置文件来修改这种加载顺序，在 weblogic.xml 中可以加入以下代码段。

示例 6-2：

```
<container-descriptor>
<prefer-web-inf-classess>true</ prefer-web-inf-classess>
</ container-descriptor>
```

上面提到用户为拓展功能可以定制自己的 ClassLoader。WebLogic 中自定义的 ClassLoader 在 Weblogic-application.xml 中描述如下所示。

示例 6-3：

```
<classloader-struts>
<module-ref>
<module-uri>ejba.jar</module-uri>
</module-ref>
<module-ref>
<module-uri>webc.war</module-uri>
</module-ref>
<classloader-structure>
<module-ref>
<module-uri>weba.war</module-uri>
</module-ref>
</classloader-structure>
<classloader-structure>
<module-ref>
<module-uri>ejbc.jar</module-uri>
</module-ref>
<module-ref>
<module-uri>webb.war</module-uri>
</module-ref>
<classloader-structure>
```

```
<module-ref>
<module-uri>webd.war</module-uri>
</module-ref>
</classloader-structure>
<classloader-structure>
<module-ref>
<module-uri>ejbb.jar</module-uri>
</module-ref>
</classloader-structure>
</classloader-structure>
</classloader-structure>
```

WebLogic 中定义的 ClassLoader 的层次结构如图 6-2 所示。

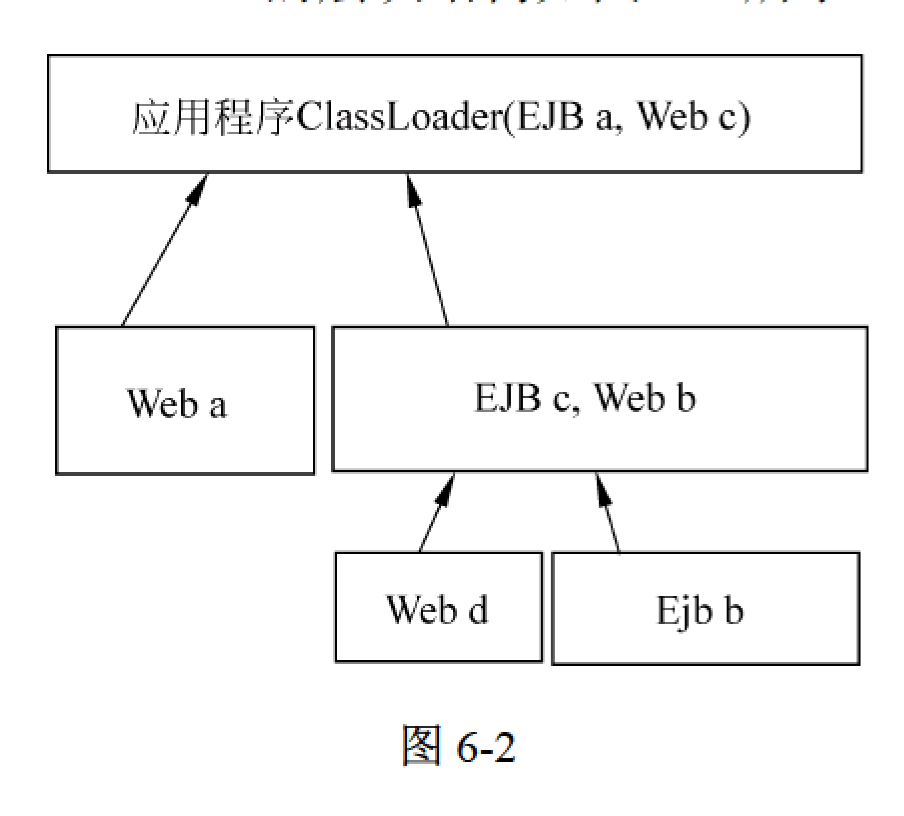


图 6-2

6.4.4　WebLogic Server 对应用程序类加载的机制

1. WebLogic Server 中应用程序类加载的概述

WebLogic Server 类加载以应用程序的概念为核心。应用程序通常打包为企业归档（EAR）文件，其中包含应用程序类。EAR 文件中的所有内容均被视为同一个应用程序的组成部分。下面的内容可以作为 EAR 的一部分，也可以作为独立应用程序进行加载。

- ❑ Enterprise JavaBean (EJB) JAR 文件。
- ❑ Web 应用程序 WAR 文件。
- ❑ 资源适配器 RAR 文件。



有关资源适配器和类加载的信息，可参阅关于资源适配器类的相关内容。

如果分别部署 EJB 和 Web 应用程序，会将它们视为两个应用程序。如果将它们一起部署在 EAR 文件中，则它们是一个应用程序。可以将模块共同部署于一个 EAR 文件内，以便系统将它们视为同一个应用程序的各部分。

每个应用程序都有其自己的类加载器层次结构，该层次结构的父级是系统类路径类加载器。它可以隔离应用程序，以使应用程序 A 无法查看应用程序 B 的类加载器或类。在层次结构类加载器中，不存在同级或同伴的概念。应用程序代码只能看到与该应用程序（或模块）关联的类加载器所加载的类，以及应用程序（或模块）类加载器的父类加载器所加载的类。这允许 WebLogic Server 在同一个 JVM 中承载多个隔离的应用程序。

2. 应用程序类加载器层次结构

部署应用程序时，WebLogic Server 自动创建类加载器的层次结构。该层次结构的根类加载器将加载应用程序中的所有 EJB JAR 文件将针对每个 Web 应用程序 WAR 文件创建子类加载器。

由于 Web 应用程序通常会调用 EJB，所以 WebLogic Server 应用程序类加载器体系结构允许 Java Server Page（JSP）文件和 Servlet 查看其父类加载器中的 EJB 接口。这种体系结构还允许在不重新部署 EJB 层的情况下重新部署 Web 应用程序。实际上，通常会更改 JSP 文件和 Servlet，而不更改 EJB 层。

图 6-3 说明此 WebLogic Server 应用程序类加载的概念。

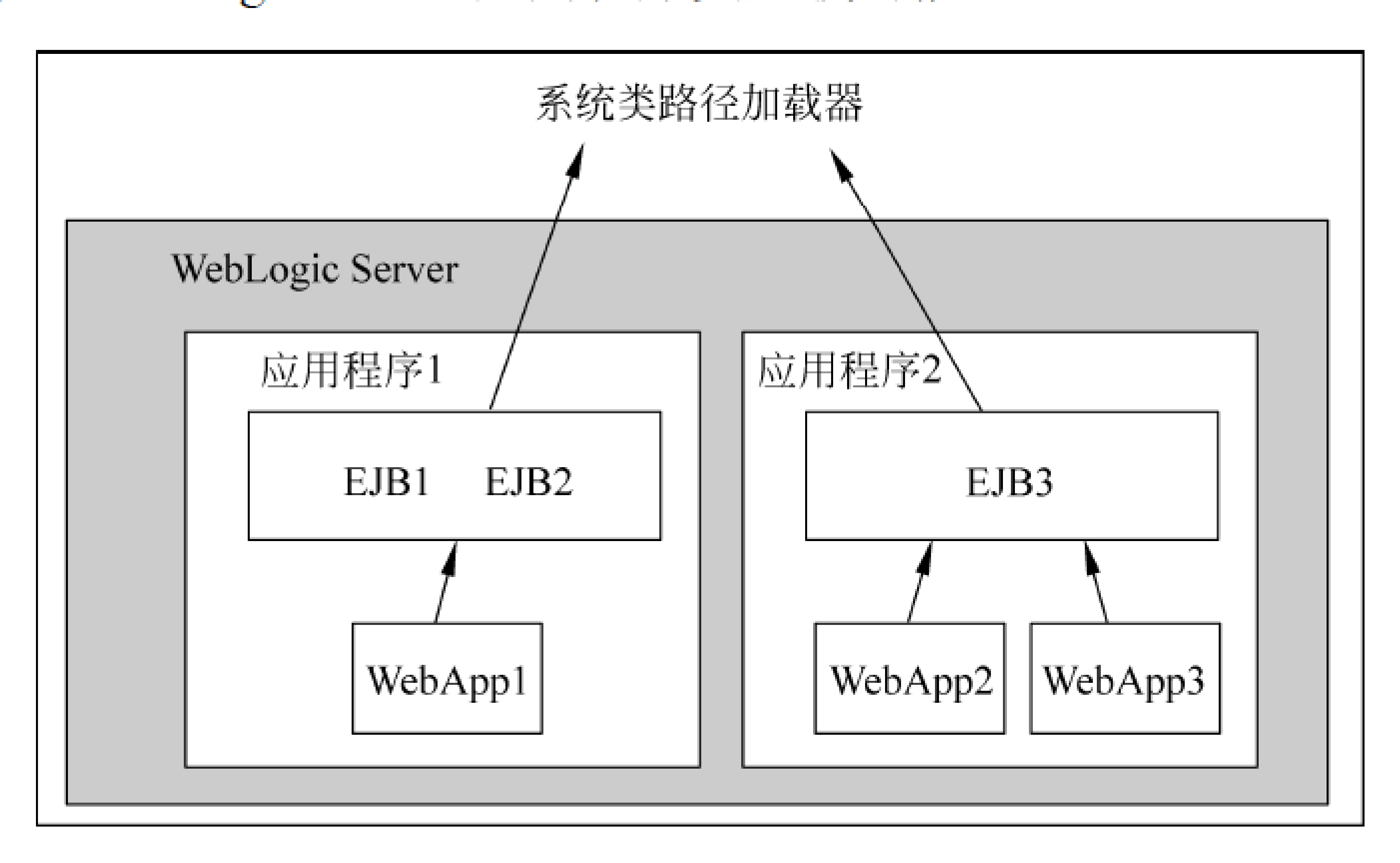


图 6-3

如果您的应用程序包含使用 EJB 的 Servlet 和 JSP：

- 将 Servlet 和 JSP 打包在 WAR 文件中。
- 将 Enterprise JavaBean 打包在 EJB JAR 文件中。
- 将 WAR 和 JAR 文件打包在 EAR 文件中。
- 部署 EAR 文件。

尽管可以分别部署 WAR 和 JAR 文件，但如果将它们共同部署于 EAR 文件内，则会生成允许 Servlet 和 JSP 查找 EJB 类的类加载器安排。如果分别部署 WAR 和 JAR 文件，WebLogic Server 将为它们创建同级类加载器。这表明您必须在 WAR 文件中包含 EJBHome 接口和远程接口，且 WebLogic Server 必须针对 EJB 调用 RMI 存根和骨架类，如同 EJB 客户端和实现类位于不同 JVM 中一样。下一部分应用程序类加载和按值传递或按引用传递中将详细讨论该概念。

注 意

Web 应用程序类加载器中包含 Web 应用程序的所有类，但 JSP 类除外。JSP 类包含其自己的类加载器，它是 Web 应用程序类加载器的子类加载器，从而允许分别重新加载各个 JSP。

3. 自定义模块类加载器层次结构

可以为应用程序创建自定义类加载器层次结构，从而更好地控制类是否可见以及是否可重新加载。可以通过在 weblogic-application.xml 部署描述符文件中定义 classloader-structure 元素来实现自定义。

图 6-4 说明如何组织 WebLogic 应用程序默认的类加载器的结构。具有应用程序级类加载器它可以加载所有 EJB 类。对于每个 Web 模块，都有适用于该模块的类的独立子类加载器。

为了简化起见，图 6-4 中不说明 JSP 类加载器。

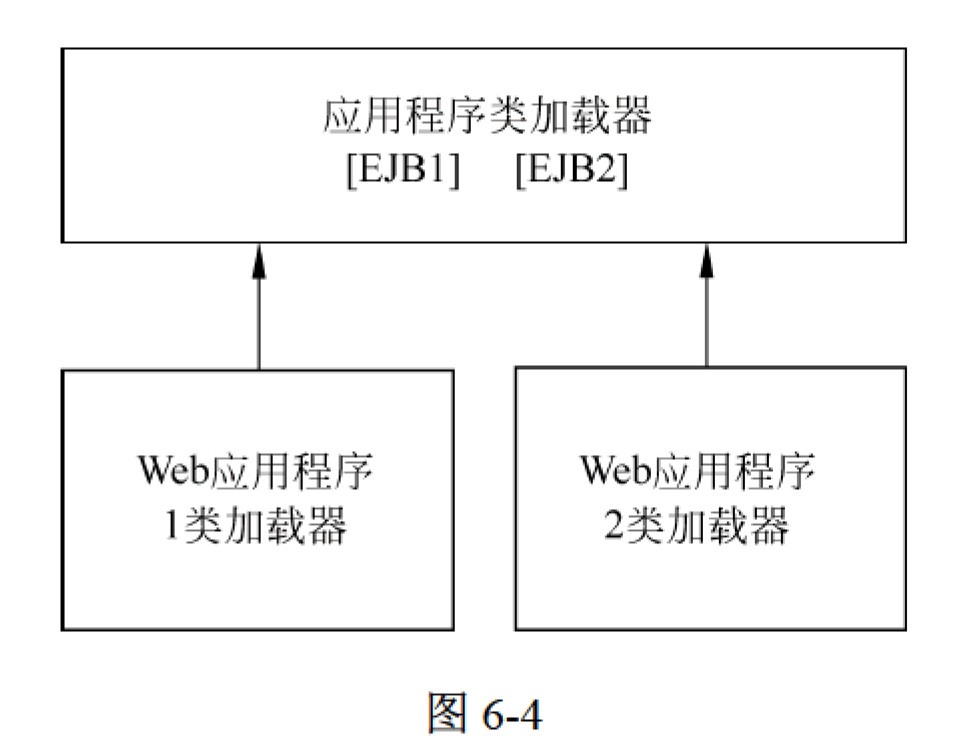


图 6-4

该层次结构对于多数应用程序都具有最佳效果，因为在调用 EJB 时，它支持使用 call-by-reference 语义。它还允许独立地重新加载 Web 模块，而不影响其他模块。而且，它允许其中一个 Web 模块中运行的代码加载任意 EJB 模块中的类。这很方便，因为 Web 模块不必包含其使用的 EJB 的接口。注意：严格来说，其中某些优点并不与 J2EE 规范严格符合。

创建自定义模块类加载器的功能提供了声明备用类加载器组织的一种机制，从而支持下列操作。

- 独立地重新加载各个 EJB 模块。
- 重新加载要一起重新加载的模块组。
- 颠倒特定 Web 模块和 EJB 模块之间的父子关系。
- EJB 模块之间的名称空间分隔。

4. 声明类加载器层次结构

可以在 WebLogic 特定的应用程序部署描述符 weblogic-application.xml 中声明类加载器的层次结构。

清单声明类加载器层次结构如下。

示例 6-4：

```
<!ELEMENT classloader-structure (module-ref*, classloader-structure*)>
<!ELEMENT module-ref (module-uri)>
<!ELEMENT module-uri (#PCDATA)>
```

weblogic-application.xml 中的顶级元素包含一个可选的 classloader-structure 元素。如果不指定该元素，则将使用标准类加载器。同时，如果某特定模块未包含于定义中，则将按照标准层次结构的定义为其分配一个类加载器。即 EJB 模块与应用程序的“根”类加载器关联，而 Web 应用程序模块具有自己的类加载器。

通过 classloader-structure 元素可以嵌套 classloader-structure 节，以便可以描述类加载器的任意层次结构。当前限制为 3 级嵌套。最外层的条目指明应用程序类加载器。对于未列出的模块，将使用标准层次结构。

注 意

此定义 scheme 中不包含 JSP 类加载器。JSP 将始终加载至其所属的 Web 模块所关联的类加载器的子类加载器中。

有关 DTD 元素的详细信息，可参考企业应用程序部署描述符元素的相关内容。

下面是一个类加载器声明（在 weblogic-application.xml 中的 classloader-structure 元素中定义）的示例。

示例 6-5：

```
<classloader-structure>
    <module-ref>
        <module-uri>ejb1.jar</module-uri>
    </module-ref>
    <module-ref>
        <module-uri>web3.war</module-uri>
    </module-ref>
    <classloader-structure>
        <module-ref>
            <module-uri>web1.war</module-uri>
        </module-ref>
    </classloader-structure>
    <classloader-structure>
        <module-ref>
            <module-uri>ejb3.jar</module-uri>
        </module-ref>
        <module-ref>
            <module-uri>web2.war</module-uri>
        </module-ref>
        <classloader-structure>
            <module-ref>
                <module-uri>web4.war</module-uri>
```

```
            </module-ref>
        </classloader-structure>
        <classloader-structure>
            <module-ref>
                <module-uri>ejb2.jar</module-uri>
            </module-ref>
        </classloader-structure>
    </classloader-structure>
</classloader-structure>
```

嵌套的组织结构指明类加载器的层次结构。以上各部分将形成如图 6-5 所示的层次结构。

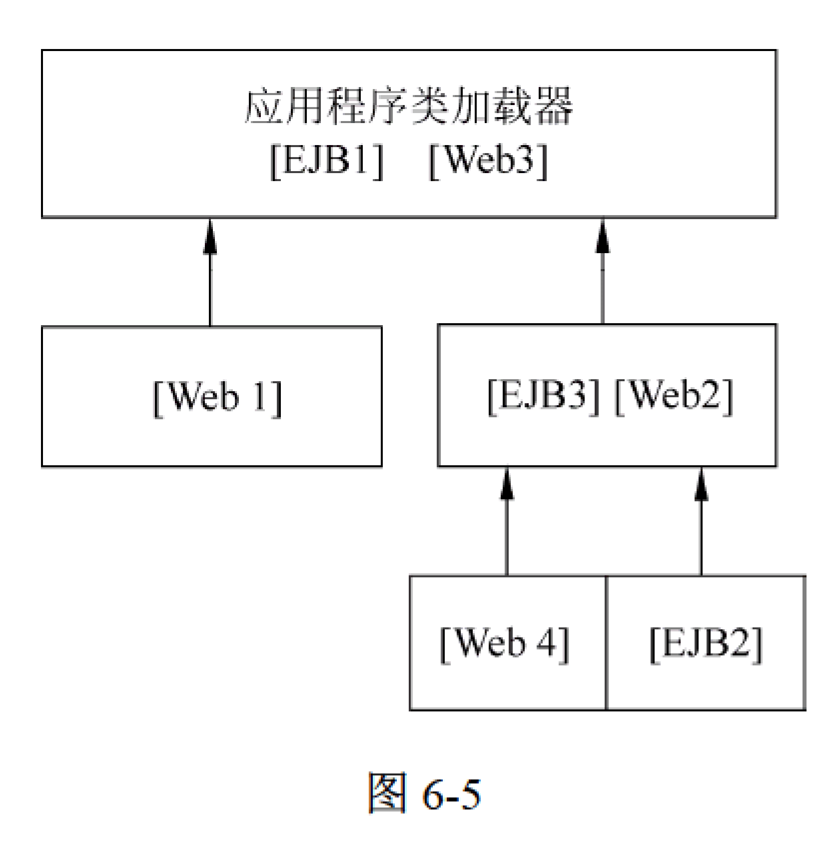


图 6-5

5. 用户定义类加载器的限制

通过用户定义的类加载器限制，可以更好地控制可重新加载的类，还可以提供模块内的类的可见性，此功能主要供开发人员使用。它适用于迭代开发，但不建议在生产应用中使用此功能的重新加载功能，因为如果更新中包含无效元素，则可能会损坏正在运行的应用程序。可以在生产中应用名称空间隔离和类可见性的自定义类加载器安排。但程序员应明确知晓，J2EE 规范要求应用程序不依赖于任何给定的类加载器组织。

某些类加载器层次结构会导致应用程序内的模块的行为，如同两个独立应用程序中的模块。例如，如果将 EJB 置于其自己的类加载器中，以便对其分别加载，则将获得 call-by-value 语义，而不是 BEA 在标准类加载器层次结构中提供的 call-by-reference 优化。同时请注意，如果您使用自定义层次结构，可能会遇到过期引用而结束。因此，如果您重新加载 EJB 模块，也应该重新加载其调用模块。

6. 实现类的单个 EJB 类加载器

WebLogic Server 允许您重新加载单独的 EJB 模块，而不要求您同时重新加载其他模块，也不必重新部署整个 EJB 模块。此功能类似于目前在 WebLogic Server Servlet 容器中重新加载 JSP 的方式。

由于 EJB 类是通过接口调用的，所以可能将单独的 EJB 实现类加载到其自己的类加载器中。这样，可以单独地重新加载这些类，而不必重新部署整个 EJB 模块。图 6-6 显示了

单个 EJB 模块的示例类加载器层次结构的外观。模块包含两个 EJB（Foo 和 Bar）。这是前一部分中描述的应用程序一般层次结构的子树。

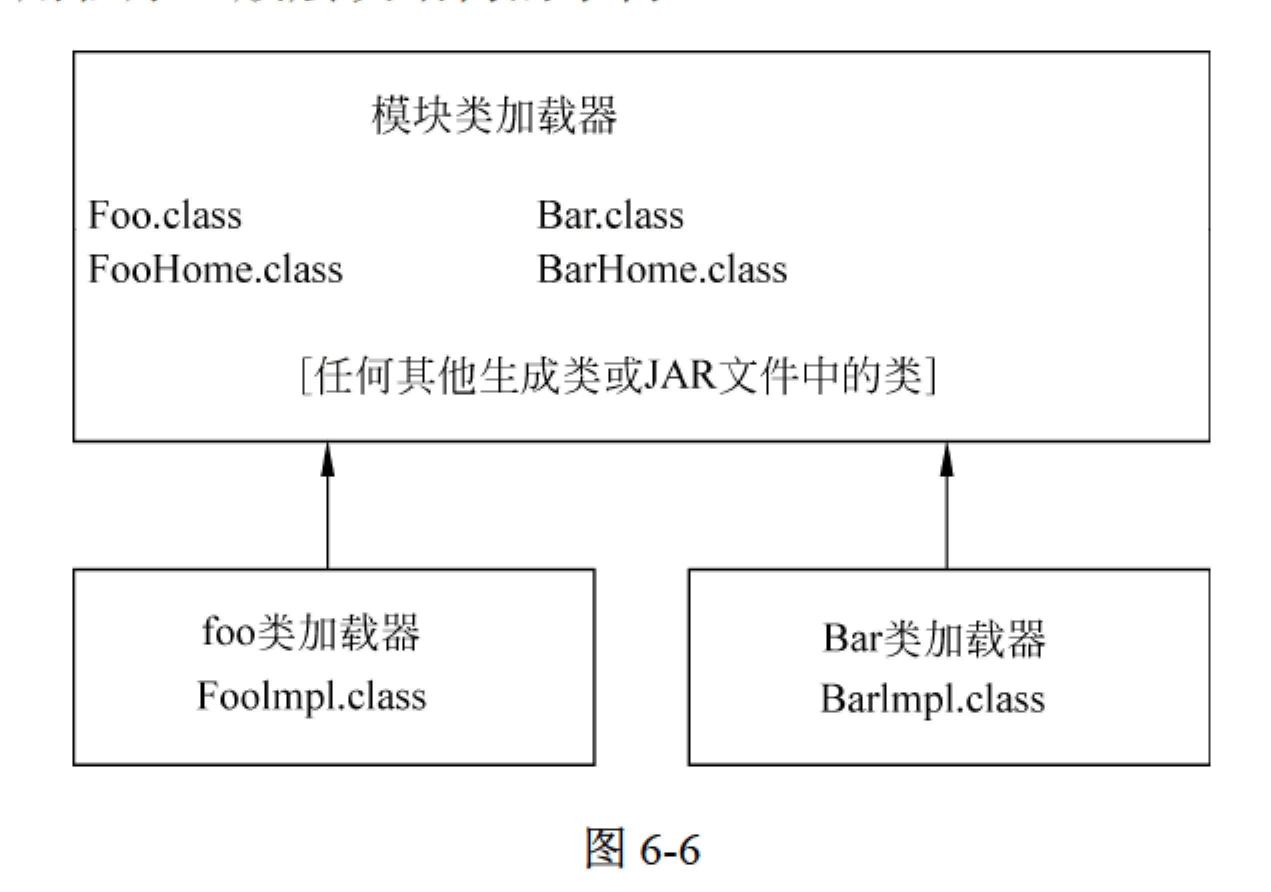


图 6-6

要对文件（相对于所展开应用程序的根）的执行部分进行更新，可使用以下命令行：

```
java weblogic.Deployer -adminurl url -user user -password password
-name myapp -redeploy myejb/foo.class
```

在-redeploy 命令后，提供要更新的文件（相对于已展开的应用程序的根）列表。这可能是特定元素（如上）或模块（或任何一组元素和模块）的路径。例如，提供要更新的相对文件的列表：

```
java weblogic.Deployer -adminurl url -user user -password password
-name myapp -redeploy mywar myejb/foo.class anotherejb
```

给定一组要更新的文件，系统将尝试计算重新部署所需要的最少内容。如果仅重新部署 EJB 实现类，则将仅重新部署该类。如果指定整个 EJB（在上例中为 anotherejb）或如果更改并更新 EJB Home 接口，则必须重新部署整个 EJB 模块。

该重新部署可能会导致其他模块也重新部署，具体取决于类加载器的层次结构。特别是，如果其他模块共享 EJB 类加载器，或加载到该 EJB 的类加载器（如 WebLogic Server 标准类加载器模块中）的子类加载器中，则这些模块也将重新加载。

7．应用程序类加载和按值传递或按引用传递

现代编程语言使用两种常见参数传递模型：按值传递和按引用传递。通过按值传递，将为每个方法调用复制参数和返回值。通过按引用传递，将向方法传递实际对象的指针（或引用）。因为按引用传递可以避免复制对象，所以它可以提高性能，但它也提供了修改被传递参数状态的一种方法。

WebLogic Server 中包含一种优化方法，能够提高服务器内远程方法接口（RMI）调用的性能。它不使用按值传递以及 RMI 子系统的编组和解组工具，而是由服务器通过按引用传递直接调用 Java 方法。此机制大大提高了性能，还可用于 EJB 2.0 本地接口。

RMI 调用优化和按引用调用仅适用于调用方和被调用方位于同一个应用程序内的情

况。同样，这也与类加载器相关。由于应用程序具有自己的类加载器层次结构，任何应用程序类都会在两个类加载器中定义，如果您尝试在应用程序之间进行分配，则会引发 ClassCastException 错误。为解决这个问题，WebLogic Server 在应用程序之间使用按值传递，即使应用程序位于同一个 JVM 中，也会使用该方法。

注 意

应用程序之间的调用比同一应用程序内部的调用慢。如果将模块统一部署为一个 EAR 文件，则可以快速调用 RMI，还支持使用 EJB 2.0 本地接口。

8．使用筛选类加载器

在 WebLogic Server 中，系统类路径中的全部.jar 文件由 WebLogic Server 系统类加载器加载。服务器实例中运行的所有应用程序都在应用程序类加载器（系统类加载器的子类加载器）中加载。在系统类加载器的实现中，应用程序不能使用系统类加载器中已有的不同版本的第三方 jar。每个子类加载器都会要求父类加载器（系统类加载器）提供特定的类，但不能加载父类加载器所见的类。

例如，如果$CLASSPATH 以及应用程序 EAR 中都包含名为 com.foo.Baz 的类，则将加载$CLASSPATH 中的类，而不加载 EAR 中的类。由于 weblogic.jar 位于$CLASSPATH 中，应用程序无法替代任何 WebLogic Server 系统的类。

以下部分定义并介绍如何使用筛选类加载器。

- 什么是筛选类加载器。
- 配置 FilteringClassLoader。
- 资源加载顺序。

（1）什么是筛选类加载器？

FilteringClassLoader 提供一种机制，可用于对部署描述符进行配置，以明确指定，特定的包应始终从应用程序中加载，而不应由系统类加载器加载。这样，您可以使用其他版本的应用程序，例如 Xerces 和 Ant。

FilteringClassLoader 位于应用程序类加载器和系统之间。它是系统类加载器的子类加载器，也是应用程序类加载器的父类加载器。FilteringClassLoader 侦听 loadClass（String className）方法，并将 className 与 weblogic-application.xml 文件中指定的一列包进行比较。如果该包匹配 className，则 FilteringClassLoader 会引发 ClassNotFoundException，该异常将通知应用程序类加载器从应用程序中加载类。

（2）配置 FilteringClassLoader。

要配置 FilteringClassLoader，从而指定应从应用程序中加载特定的包，需要向 weblogic-application.xml 文件中添加 prefer-application-packages 描述符元素，该文件详细列出要从应用程序中加载的包。下列示例指定 org.apache.log4j.*和 antlr.*包应从应用程序中加载而不从系统加载器中加载。

示例 6-6：

```
<prefer-application-packages>
  <package-name>org.apache.log4j.*</package-name>
```

```
    <package-name>antlr.*</package-name>
  </prefer-application-packages>
```

（3）资源加载顺序。

资源加载顺序是 java.lang.ClassLoader 的 getResource()方法和 getResources()方法返回资源的顺序。启用筛选时，该顺序与禁用筛选时的顺序稍有不同。启用筛选表示 FilteringClassLoader 中包含一个或多个包模式。不使用任何筛选（默认）时，将按照类加载器树的自上而下的顺序收集资源。例如，如果 Web（1）请求资源，资源分组的顺序为系统（3）、应用程序（2）和 Web（1）。

使用系统类加载器：

```
系统(3)
|
应用程序 (2)
|
Web (1)
```

为使说明更明确，给定一个资源 /META-INF/foo.xml，它位于所有类加载器中且会返回下列 URL。

示例 6-7：

```
META-INF/foo.xml - from the System ClassLoader (3)
META-INF/foo.xml - from the App ClassLoader (2)
META-INF/foo.xml - from the Web ClassLoader (1)
```

启用筛选时，将返回从 FilteringClassLoader（应用程序类加载器）的子类加载器到该调用类加载器中的资源，然后再返回系统类加载器中的资源。在图 6-6 中，如果同一资源位于所有类加载器（D）、（B）和（A）中，当 Web 类加载器请求该资源时，则将以如下顺序获得这些资源。

示例 6-8：

```
META-INF/foo.xml - from the App ClassLoader (B)
META-INF/foo.xml - from the Web ClassLoader (A)
META-INF/foo.xml - from the System ClassLoader (D)
```

> **注 意**
>
> 返回资源时将依据 FilteringClassLoader 下的默认 J2EE 委托模型，只将 FilteringClassLoader 的父类加载器中的资源追加到所返回枚举值的末尾。

使用筛选类加载实现：

```
系统(D)
|
FilteringClassLoader (filterList := x.y.*) (C)
|
```

```
应用程序 (B)
|
Web (A)
```

如果应用程序类加载器请求同一资源，将获得以下顺序。

示例 6-9：

```
META-INF/foo.xml - from the App ClassLoader (B)
META-INF/foo.xml - from the System ClassLoader (D)
```

getResource() 只返回第一个描述符，getResourceAsStream() 返回第一个资源的 InputStream。

第 3 篇

实　施　篇

第 7 章　集群的安装与配置

7.1　集群知识回顾

7.1.1　集群概念

WebLogic Server 集群包含多个 WebLogic Server 服务器实例，每个实例同时运行并协同工作，以提供更高的性能、可用性和稳定性。集群对客户端来讲就像一个虚拟的 WebLogic Server 实例。由服务器实例组成的集群可以运行于同一台计算机上，也可以被部署在不同的计算机上。集群中的每个服务器实例要求运行相同版本的 WebLogic Server。

7.1.2　集群的体系结构

在集群中所有服务器实例必须在同一域中，属于不同域的服务器实例不可以属于一个集群。

由于一个域中只会有一个管理服务器，如果一个域包含多个集群，所有的集群在域中都有同一个管理服务器统一管理。

集群的 WebLogic Server 实例的行为类似于非集群实例，但它们支持故障转移和负载平衡。通常的集群体系结构如图 7-1 所示。

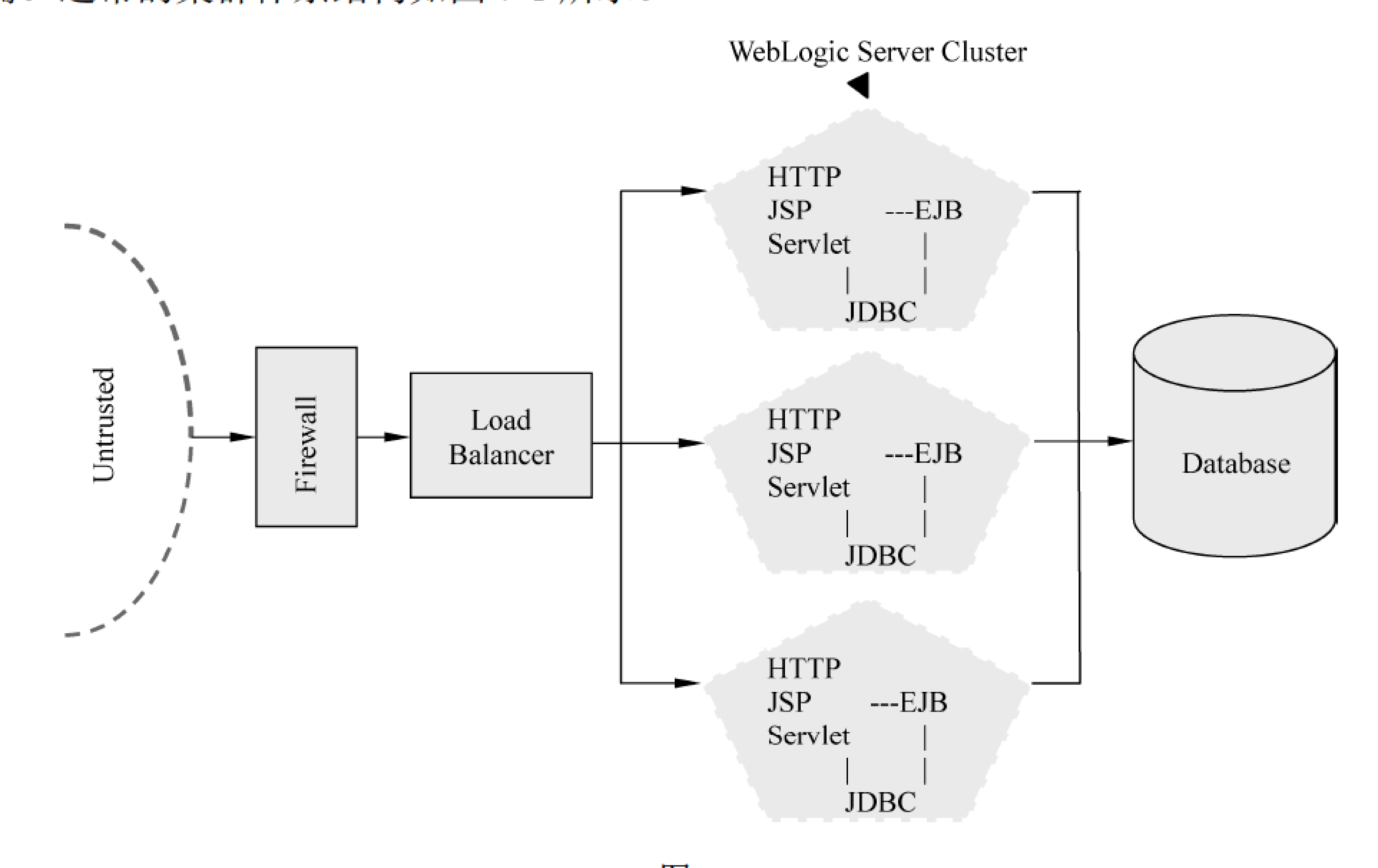


图 7-1

7.2　创建集群的条件

配置集群之前，需要执行以下步骤来准备环境。

1．获取集群许可证（此步骤限于 9.2 及之前的版本）

集群 WebLogic Server 实例必须具有有效的集群许可证。

要更新当前的许可证时，可使用 BEA 主目录中的 UpdateLicense 工具：

```
UpdateLicense.cmd。
- UpdateLicense <new_license_file>
```

请确保 license 文件中包含有效的对“Cluster”组件的授权，其内容类似如下的部分。

示例 7-1：

```
<license
component="Cluster"
cpus="unvalued"
expiration="never"
ip="any"
licensee="BEA Internal Development"
serial="616351266349-1844896394531"
type="SDK"
units="5"
signature="MC0CFQCQrk+Kbddfz3RHVH6uGfj"
/>
```

2．了解网络和安全拓扑

（1）集群是否位于单个局域网中？

（2）集群是跨 LAN 还是 WAN？

根据您选择的网络拓扑，安全要求也将会有所不同。某些网络拓扑会干扰多播通信，所以请尽量避免跨防火墙部署集群中的服务器实例。

3．确定集群架构

（1）使用单层架构还是多层架构？

（2）计划如何执行负载均衡？

（3）是否要使用基本的 WebLogic Server 负载均衡？

（4）是否要使用第三方负载均衡器？

（5）是否将隔离区与防火墙配合使用？

您所选择的架构将对集群的设置方式产生影响。根据集群架构，您可能还需要安装或配置其他资源，如负载均衡器、HTTP 服务器和代理插件。

4. 选择要进行集群安装的计算机

（1）可以在单台计算机上设置集群来进行演示或开发，不过这对生产环境并不实用。

（2）计算机不要使用动态分配的 IP 地址。

（3）理论上对在集群中的服务器实例数量没有限制，只要有合适的许可证（License）。

（4）大型多处理器服务器可以承载大型集群，一般建议每两个 CPU 对应一个 WebLogic Server 实例（当然具体还需要根据应用的负载模型来确定）。

集群的主要优点是负载平衡和故障转移。如果集群中的多个服务器位于同一台计算机上，则这些优点将显现不了。如果计算机出现故障，位于此计算机上的所有服务器也都将出现故障，即使负载平衡，处理过程也只能由该计算机进行。负载平衡器和代理服务器需要了解哪些服务器位于一个集群中，因此，一般情况下，您需要在负载平衡或代理服务器中配置集群中每个服务器的 IP 地址。如果将服务器分配给动态分配 IP 地址的计算机，那么 IP 地址会变化，负载平衡器或者代理服务器将无法找到它。

5. 确定集群中服务器实例的 IP 地址或 DNS 名称以及端口号

在程序中调用实体 Bean 和会话 Bean 时，建议使用集群地址作为 Provider_URL 来构造请求，并且在集群地址中使用 DNS 名称，此名称可通过 DNS 映射至集群中每个 WebLogic Server 实例的 IP 地址。

动态集群地址需要符合以下格式（以集群中有三个实例为例）：

```
listenaddress1:listenport1,listenaddress2:listenport2,listenaddress3:
listenport3
```

7.3 代理服务器 Proxy

7.3.1 代理服务的角色和作用

代理插件提供了以下优点。

1. 利用现有的硬件

如果您已经有一个 Web 服务器（Web Server，一般用于提供静态内容），您可以复用现有的 Web 服务器，为部署在后端 WebLogic 上的应用请求提供动态的 HTTP 负载均衡和故障恢复。

2. 熟悉防火墙策略

使用 Web 服务器代理使您能够使用熟悉的防火墙政策，以确定您的 DMZ policy。在一般情况下，您可以继续在 DMZ 区域放置 Web 服务器，而不允许客户端直接连接到集群内的 WebLogic 服务器上。

3．错误恢复

简单而言，failover 的意思是当一个执行项特定工作的应用组件/服务因为某种原因而变得不可用时，一个该组件的备份可以继续完成该任务。

WebLogic Server 使用标准的通信技术和工具，比如多播（Multicast）、IP Sockets 和 JNDI（Java Naming and Directory Interface）来共享和维护集群中对象的可用性信息。这些技术使得 WebLogic Server 可以检测对象在未完成其任务之前就停止的错误，并调度另外一个对象的备份来完成剩余的任务。

关于一项工作完成状态（完成了哪些工作）的信息叫做状态。WebLogic Server 维护状态信息的技术包括会话复制和 replica-aware 存根。当一个特定的对象非正常终止其工作时，复制技术激活该对象的一个备份，并从该对象停止处继续运行，并完成工作。

4．负载均衡

负载均衡是在计算和网络环境中对任务的分配和互相通信。负载均衡可能出现在以下情况下。

- 有多个对象可以处理相同的任务。
- 有关所有对象的位置和运行状态的信息。

WebLogic Server 允许对象被集群（在多个服务器实例上部署），所以有了多个对象可以做同一工作。

代理服务器的类型有以下几种。

基于软件的代理服务器可以是内部 WebLogic Servlet 或第三方应用程序。

基于硬件的代理服务器通常是物理负载平衡器。

7.3.2 代理服务器的配置

1．代理服务器的配置

（1）通过 WebLogic Wizard 来配置。

用 Domain Configuration Wizard 创建新 WebLogic 域时可以对其进行配置。在向导中创建集群后，将显示 Create HTTP Proxy Applications（创建 HTTP 代理应用程序）选项。未定位到集群的服务器都是 HTTP 代理服务器的候选对象。选择 Create HTTP proxy for <cluster>（为<cluster>创建 HTTP 代理）选项以及将承载此代理应用程序的服务器。

（2）手动创建 WebLogic 代理服务器。

首先在代理服务器的默认 Web 应用程序的 web.xml 文件中配置 HttpClusterServlet。此文件位于 Web 应用程序目录的\WEB-INF 目录下。

要配置 HttpClusterServlet，可执行以下操作。

① 配置一个 WebLogic Server 实例，以其作为代理将请求转到 WebLogic Server 实例的集群中。

a. 在管理控制台中创建服务器实例。

b. 将默认 Web 应用程序部署到此 WebLogic Server 实例。

② 在已部署到代理服务器上的默认 Web 应用程序的 web.xml 文件中注册 HttpClusterServlet。

HttpClusterServlet 的完整类名如下。

WLS 6.1: weblogic.servlet.internal.HttpClusterServlet

WLS 7.0, 8.1: weblogic.servlet.proxy.HttpClusterServlet

然后使用 web.xml 部署描述符中的<init-param> 元素为 HttpClusterServlet 定义适当的初始化参数。

示例 7-2：

```
<servlet>
<servlet-name>HttpClusterServlet</servlet-name>
<servlet-class>
weblogic.servlet.proxy.HttpClusterServlet
</servlet-class>
<init-param>
<param-name>WebLogicCluster</param-name>
<param-value>
serverA:7001:7002|serverB:7001:7002|serverC:7001:7002
</param-value>
</init-param>
<init-param>
<param-name>DebugConfigInfo</param-name>
<param-value>ON</param-value>
</init-param>
</servlet>
```

代理 Servlet 需要被定义为受管服务器的默认 Web 应用程序。这可以在 Web 应用程序目录的\WEB-INF 目录下的 weblogic.xml 部署描述符中定义。Servlet 映射如下。

配置 Servlet 映射。

示例 7-3：

```
<servlet>
<servlet-name>HttpClusterServlet</servlet-name>
…
</servlet>
<servlet-mapping>
<servlet-name>HttpClusterServlet</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>HttpClusterServlet</servlet-name>
<url-pattern>*.jsp</url-pattern>
</servlet-mapping>
```

将代理 Servlet 映射到<url-pattern>。具体而言，就是映射所需代理的文件的扩展名，例如*.jsp。如果将<url-pattern>设置为“/”，则任何代理服务器无法解析的请求都将被发送

到集群中的服务器。但是，如果您希望代理对*.jsp 类型文件的请求，则仍必须专门映射该文件扩展名。

2. 第三方代理服务器

如果您使用的是受支持的第三方 Web 服务器，而不是利用 WebLogic Server 作为 Web 服务器，则需要设置一个代理插件。

以下是支持的第三方 Web 服务器类型。

（1）Netscape Enterprise Server。

（2）Apache Web Server。

（3）Microsoft Internet Information Server。

7.3.3 F5 硬件负载平衡器及其他

F5 负载均衡技术

F5 BIG-IP LTM（本地流量管理器）是一台对流量和内容进行管理分配的设备。它提供 12 种灵活的算法将数据流有效地转发到它所连接的服务器集群中。而从用户角度看到的只是一台虚拟服务器。用户此时只需访问定义于 BIG-IP LTM 上的一台服务器，即虚拟服务器（Virtual Server）。但它们的数据流却被 BIG-IP 灵活地均衡分布到所有的物理服务器中。

BIG-IP LTM 可以通过多种负载均衡算法对流量进行分配，这些算法包括以下各个方面。

（1）轮询（RoundRobin）。

（2）比率（Ratio）。

（3）优先权（Priority）。

（4）最少的连接方式（LeastConnection）。

（5）最快模式（Fastest）。

（6）观察模式（Observed）。

（7）预测模式（Predictive）。

（8）动态性能分配（DynamicRatio-APM）。

（9）动态服务器补充（DynamicServerAct）。

（10）服务质量（QoS）。

（11）服务类型（ToS）。

（12）规则模式。

关于 F5 BIG-IP 的详细信息，请参考其官方文档。

7.4 如何创建集群

7.4.1 集群环境确定

集群环境确定见表 7-1。

表 7-1

Server name	Ip	Port	备注
Ms1	192.168.0.139	7001	管理服务器
As1	192.168.0.139	7002	本机被管服务器
As2	192.168.0.140	7003	远程被管服务器
Cs	239.192.0.0	7777	多播

7.4.2 集群配置步骤

图形化界面的配置比较简单，这里不做介绍，下面主要介绍以 Linux 下的字符界面配置集群。

（1）成功安装完 WebLogic 后，转到安装目录下的%weblogic_home%/wlserver_10.3/common/bin，运行 config.sh 文件，注意模式为 console，如图 7-2 所示。



图 7-2

（2）进入安装第一步，选择是新建域还是扩展现有域，我们这就从创建域开始，当然，当您已经拥有一个域时，可以选择扩展现有域。

下面我介绍一下安装过程中输入的合法性，如果提示是有选择性的，当然一般是一个数字，当您选择了相应的数值，界面的指示也会有相应的显示，图 7-3 所示默认的是选择了 1。选择确定后可以输入“Next”或者“n”进入下一步设置。

（3）接下来是选择域模板，这个模板的作用是定制您要配置哪些组件，比如，如果您就是一个单机，没有必要用集群，您就可以定制自己的配置模板，在配置过程中不显示配置集群这一步。如果想定制自己的配置过程，您就可以用到 WebLogic 提供的自定义模板的功能。这里就不做介绍了。

WebLogic 提供了一个通用的配置模板，简单的集群配置可以通过这个完成，所以可以

在默认模板下开始集群配置，如图 7-4 所示。



图 7-3

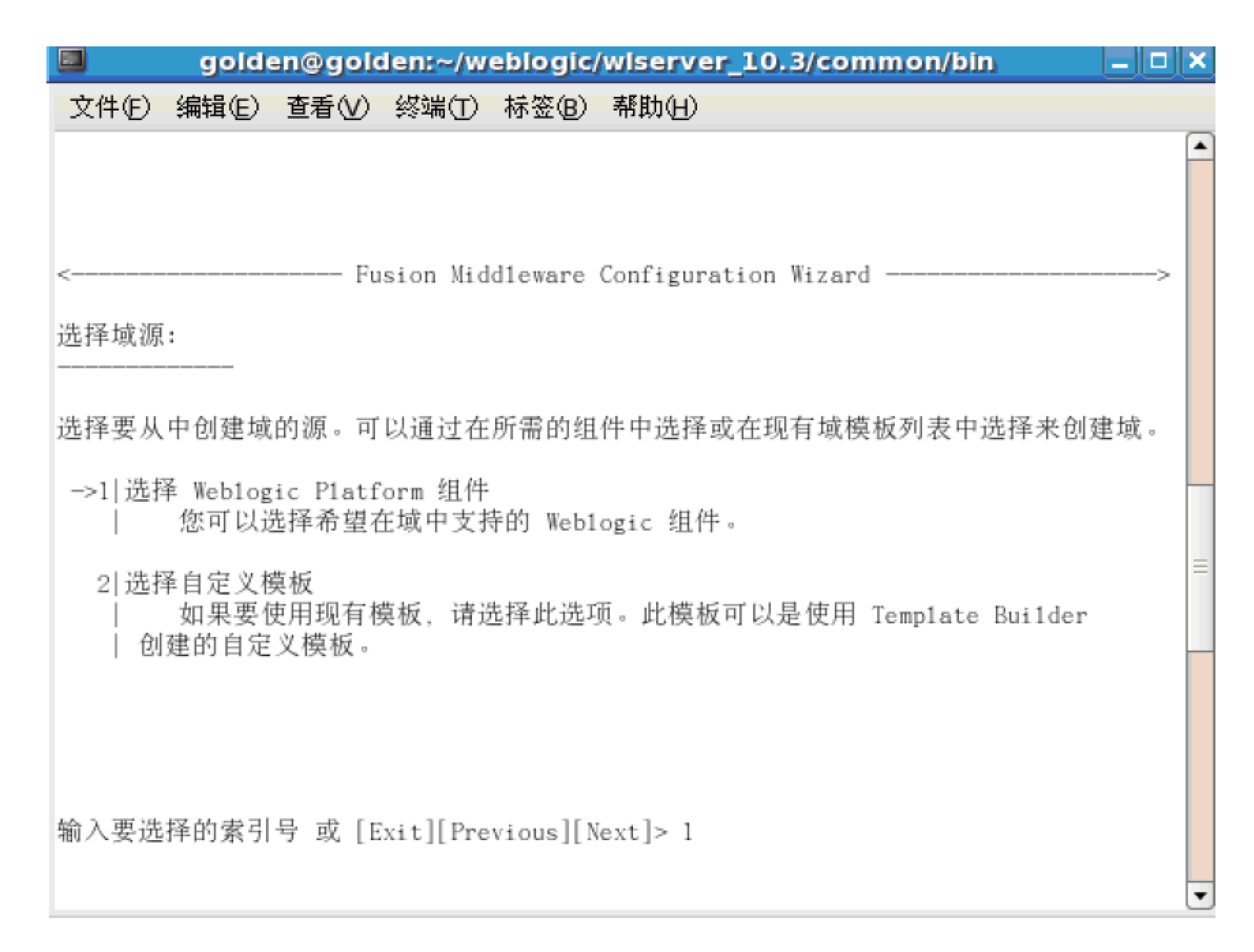


图 7-4

（4）选择了默认模板后，就会显示可用模板，这里直接进行下一步操作就可以了，如图 7-5 所示。



图 7-5

（5）接下来是配置 doman 域了，输入您的域名后按 Enter 键，再单击 Next 按钮，如图 7-6 所示。



图 7-6

（6）这一步是选择您将域安装在哪里，这里选择默认位置，如图 7-7 所示。

（7）配置 manager，注意密码至少 8 位，但不能是单一的数字或者字符，如图 7-8 所示。

（8）选择域启用的模式，有开发模式和生产模式，图 7-9 所示是两种模式的区别。



图 7-7

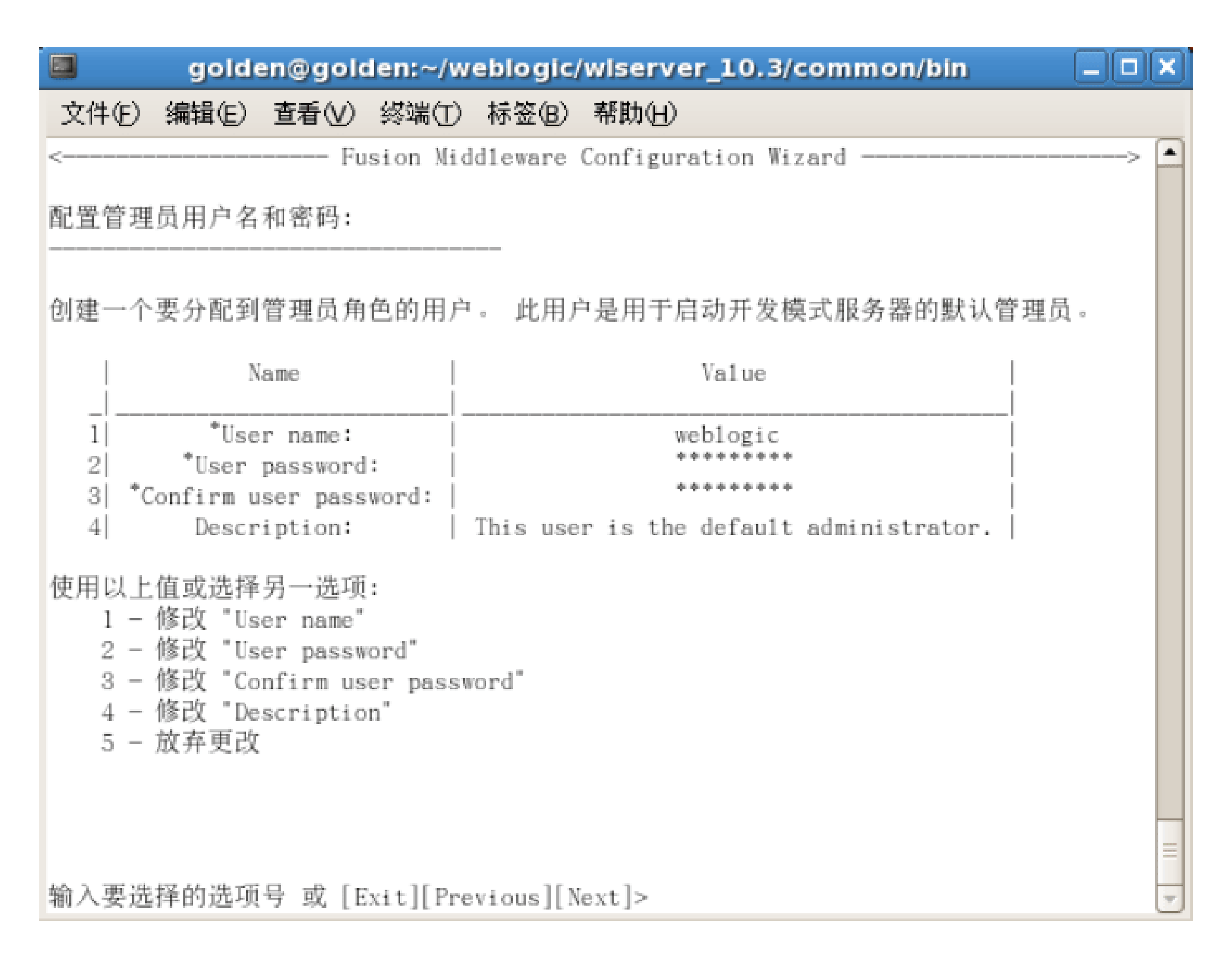


图 7-8

调整参数	开发模式	生产模式
SSL	可以使用 WebLogic Server 安全服务提供的示范数字证书和示范密钥库。利用这些证书，可设计出在由 SSL 担保的环境中工作的应用程序	不应使用示范数字证书和示范密钥库。如果这样做，将会显示警告消息
部署应用程序	WebLogic Server 实例可以自动部署和更新驻留在 domain_name/autodeploy 目录中的应用程序（其中 domain_name 为域名）。建议只在单服务器开发环境中使用此方法	由于自动部署功能已禁用，因此必须使用 WebLogic Server 管理控制台、weblogic.Deployer 工具或 WebLogic 脚本工具（WLST）
并发	默认 5 个	并发数高

图 7-9

下面为了测试，选择的是生产模式，如图 7-10 所示。



图 7-10

（9）选择 JDK，这里要特别说明的是如果是开发环境，您可以选择自己的 JDK，当是生产环境时，建议使用 WebLogic 提供的 JDK，还有 JDK 的版本要注意与 WebLogic 版本控制一致，WebLogic10.3 使用的 JDK 版本是 1.6。

这里选择自己的 JDK，如图 7-11 所示。



图 7-11

（10）接下来是选择要配置的项目，本节主要介绍配置集群，所以选择配置 1 管理服务器，2 受管服务器、集群和计算机，只要输入前面的编号，相应的项目就会被选定，如图 7-12 所示。

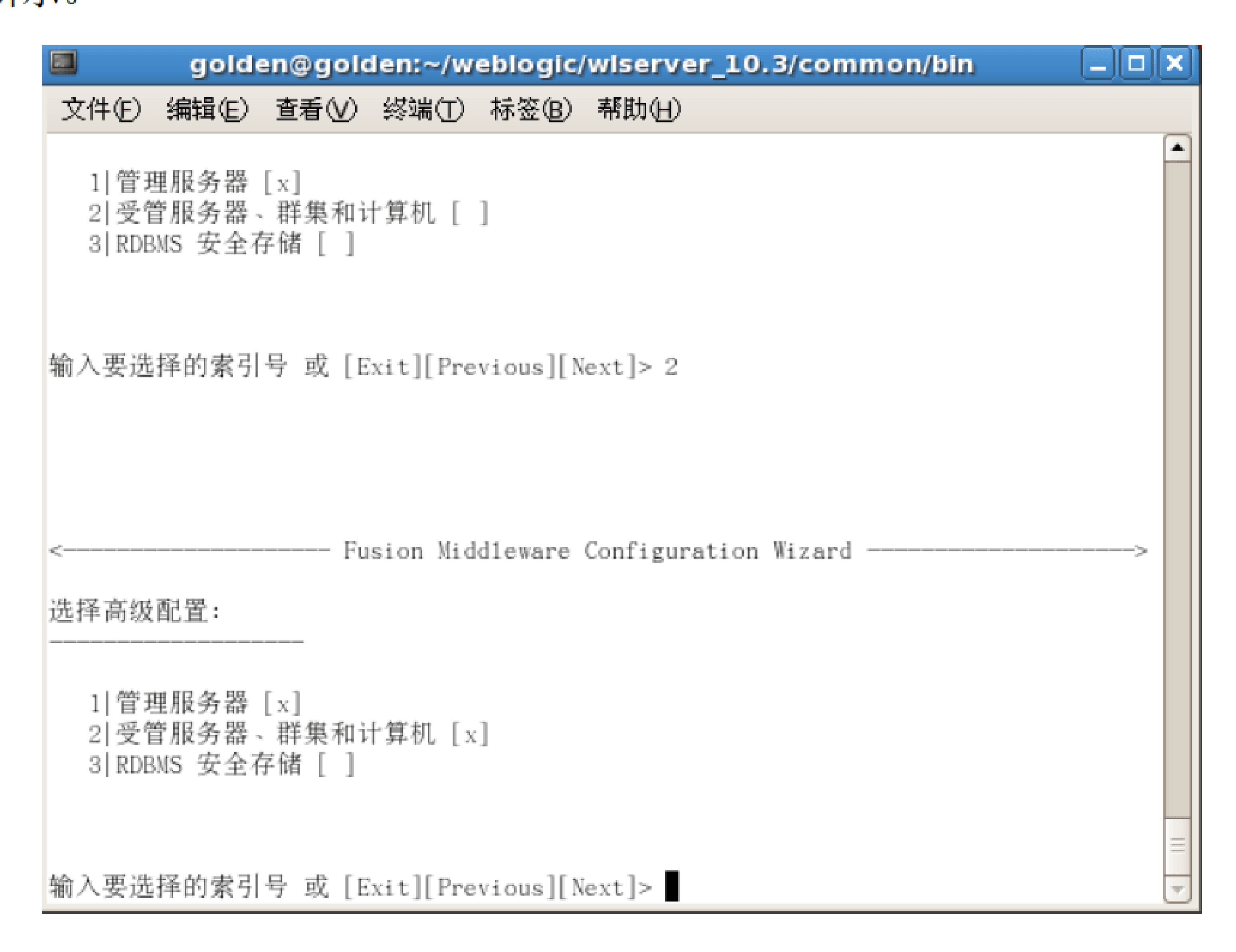


图 7-12

（11）接下来配置管理服务器，按照前面表的要求，输入正确的监听地址、端口，确定无误后单击 Next 按钮，如图 7-13 所示。

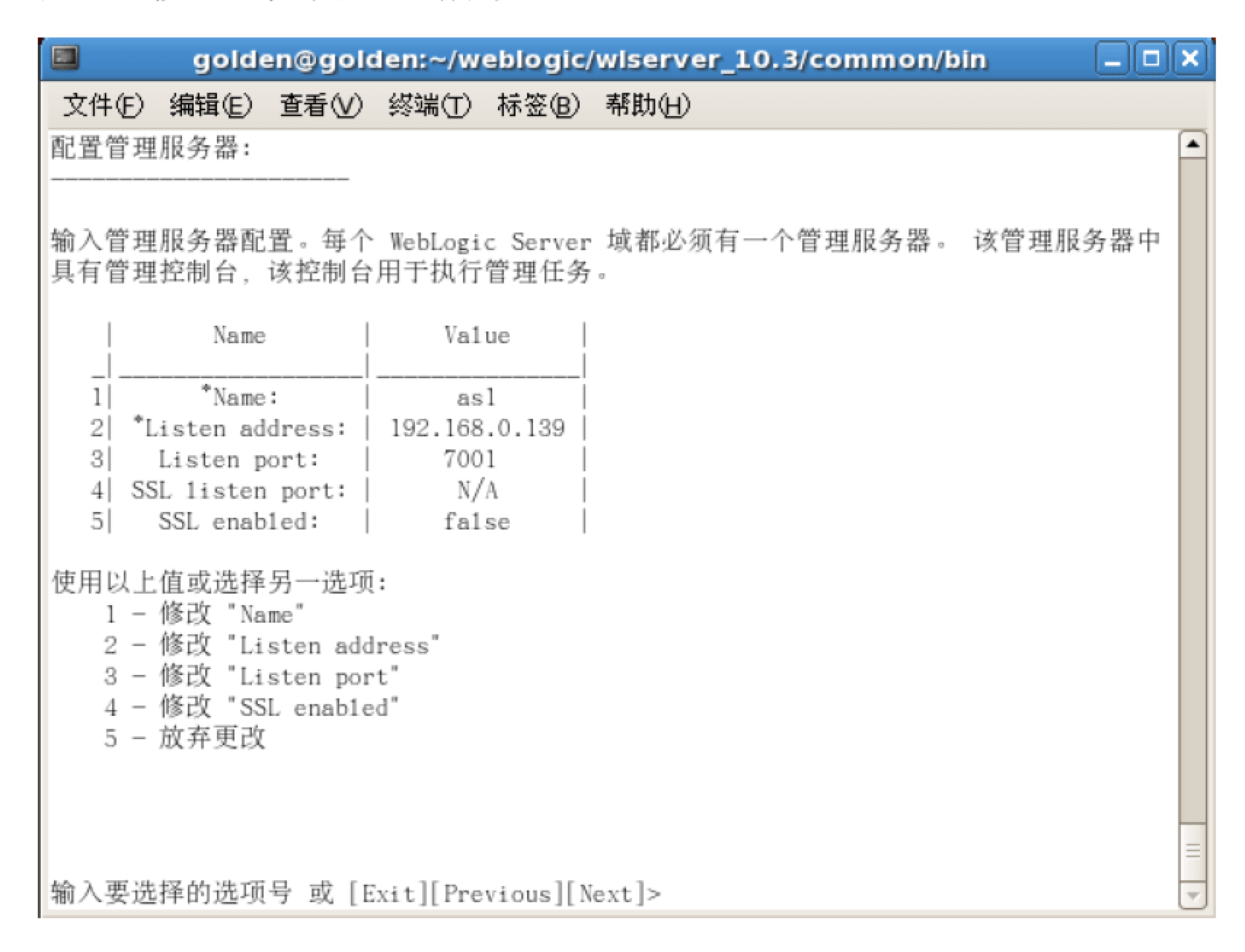


图 7-13

（12）配置受管服务器，先输入受管服务器的名称，如图 7-14 所示。



```
golden@golden:~/weblogic/wlserver_10.3/common/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)

输入要选择的选项号 或 [Exit][Previous][Next]> n

<------------------- Fusion Middleware Configuration Wizard ------------------->

配置受管服务器:
----------------------

为受管服务器添加或删除配置信息。
典型的生产环境都具有一个或多个受管服务器。 每个受管服务器都是用于承载企业应用程序的
 WebLogic Server 的实例。

   | Name* | Listen address* | Listen port | SSL listen port | SSL enabled |
  _|_______|_________________|_____________|_________________|_____________|

输入名称用于新  或 [Exit][Previous][Next]> ms1
```

图 7-14

（13）修改相应的项目，确定无误后选择“5-完成”选项，然后用同样的步骤配置受管服务器 ms2，如图 7-15 所示。



```
golden@golden:~/weblogic/wlserver_10.3/common/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)

<------------------- Fusion Middleware Configuration Wizard ------------------->

配置受管服务器:
----------------------

为受管服务器添加或删除配置信息。
典型的生产环境都具有一个或多个受管服务器。 每个受管服务器都是用于承载企业应用程序的
 WebLogic Server 的实例。

   | Name* | Listen address* | Listen port | SSL listen port | SSL enabled |
  _|_______|_________________|_____________|_________________|_____________|
1->|  ms1  |  192.168.0.139  |    7002     |       N/A       |    false    |

使用以上值或选择另一选项:
    1 - 修改 "Name"
    2 - 修改 "Listen address"
    3 - 修改 "Listen port"
    4 - 修改 "SSL enabled"
    5 - 完成

输入要选择的选项号 或 [Exit][Previous][Next]>
```

图 7-15

（14）配置 cluster server，注意选择 cluster messaging mode 的 multicast 选项，如图 7-16 所示。



图 7-16

（15）为集群分配受管服务器，如图 7-17 所示。



图 7-17

（16）这里两个都选上，如图 7-18 所示。

这里简单补充 ms2 在远程服务器的配置过程。

① 安装同一个版本的 WebLogic。

② 配置与管理服务器上同名的 Doman 域名以及 manager 口令。

③ 配置受管服务器，和主机上配置的 ms2 监听地址和端口一致。

④ 其他的集群就不用配置了。



图 7-18

到这里，一个简单的集群就配置好了，接下来是测试集群配置是否成功。

7.5　集群的启动

7.5.1　管理服务器 Admin Server 的启动

首先启动管理服务器，如图 7-19 所示。

```
golden@golden:~/weblogic/user_projects/domains/golden/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
deployments  jdbc        lib  security
[golden@golden config]$ cd startup/
[golden@golden startup]$ ls
readme.txt
[golden@golden startup]$ cd ..
[golden@golden config]$ ls
config.xml   diagnostics  jms  nodemanager  startup
deployments  jdbc         lib  security
[golden@golden config]$ cd ,,
bash: cd: ,,: 没有那个文件或目录
[golden@golden config]$ ls
config.xml   diagnostics  jms  nodemanager  startup
deployments  jdbc         lib  security
[golden@golden config]$ cd  ..
[golden@golden golden]$ ls
autodeploy  console-ext          lib       startManagedWebLogic_readme.txt
bin         fileRealm.properties security  startWebLogic.sh
config      init-info            servers
[golden@golden golden]$ cd bin/
[golden@golden bin]$ ls
nodemanager        setDomainEnv.sh           startWebLogic.sh
server_migration   startManagedWebLogic.sh   stopManagedWebLogic.sh
service_migration  startPointBaseConsole.sh  stopWebLogic.sh
[golden@golden bin]$ . ./startWebLogic.sh
```

图 7-19

管理服务器启动成功，如图 7-20 所示。



图 7-20

7.5.2　受管服务器 Managed Server 的启动

注意格式，第一个参数是输入启动脚本，第二个是输入您要启动的受管服务器名，第三个是输入您刚配置的管理服务器的地址，如图 7-21 所示。

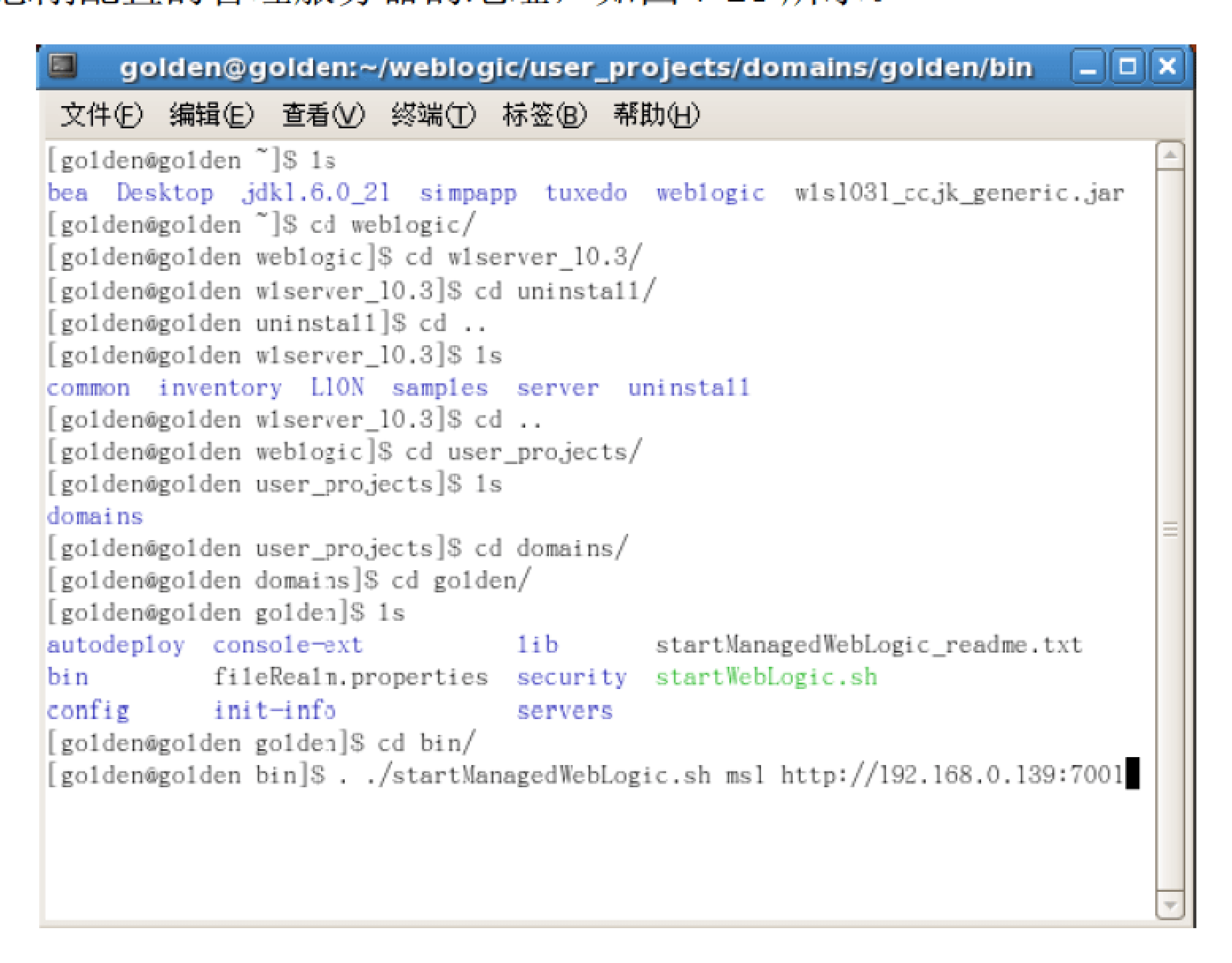


图 7-21

受管服务器启动成功，如图 7-22 所示。

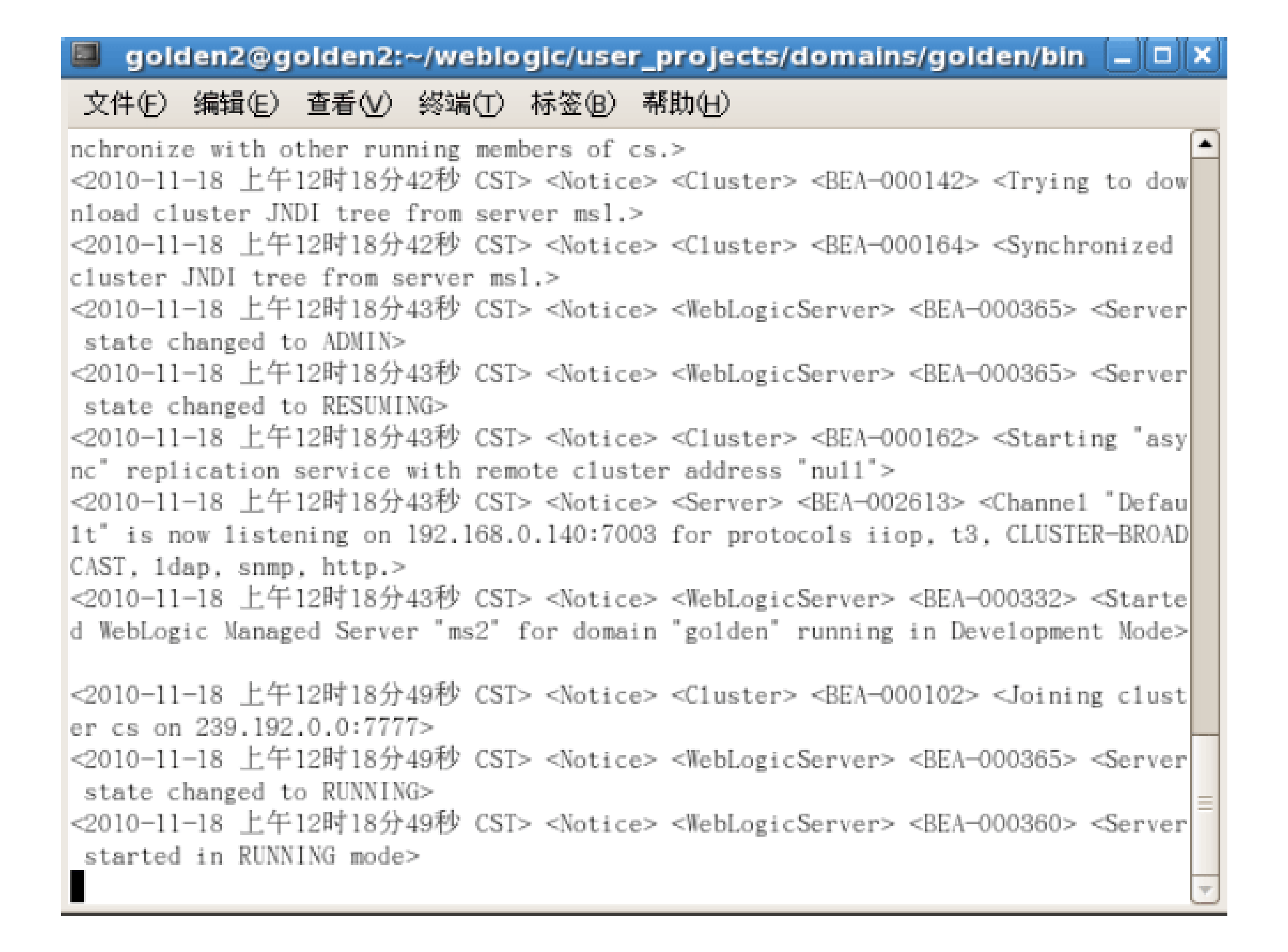


图 7-22

如果启动远程受管服务器，就不用启动管理服务器了，如图 7-23 所示。

```
golden2@golden2:~/weblogic/user_projects/domains/golden/bin
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
[golden2@golden2 ~]$ ls
bea  Desktop  jdk1.6.0_21  simpapp  tuxedo  weblogic
[golden2@golden2 ~]$ cd weblogic/
[golden2@golden2 weblogic]$ cd user_projects/
[golden2@golden2 user_projects]$ cd domains/
[golden2@golden2 domains]$ cd golden/
[golden2@golden2 golden]$ cd bin/
[golden2@golden2 bin]$ . ./startManagedWebLogic.sh ms2 http://192.168.0.139:7001
```

图 7-23

远程受管服务器启动成功，如图 7-24 所示。



图 7-24

到这一步管理服务器和受管服务器都启动成功，表明配置没有问题了。

7.6　集群中应用的部署

（1）访问控制台，选择左侧的部署选项，如图 7-25 所示。

（2）单击“安装”按钮，选择要部署的应用，如图 7-26 所示。

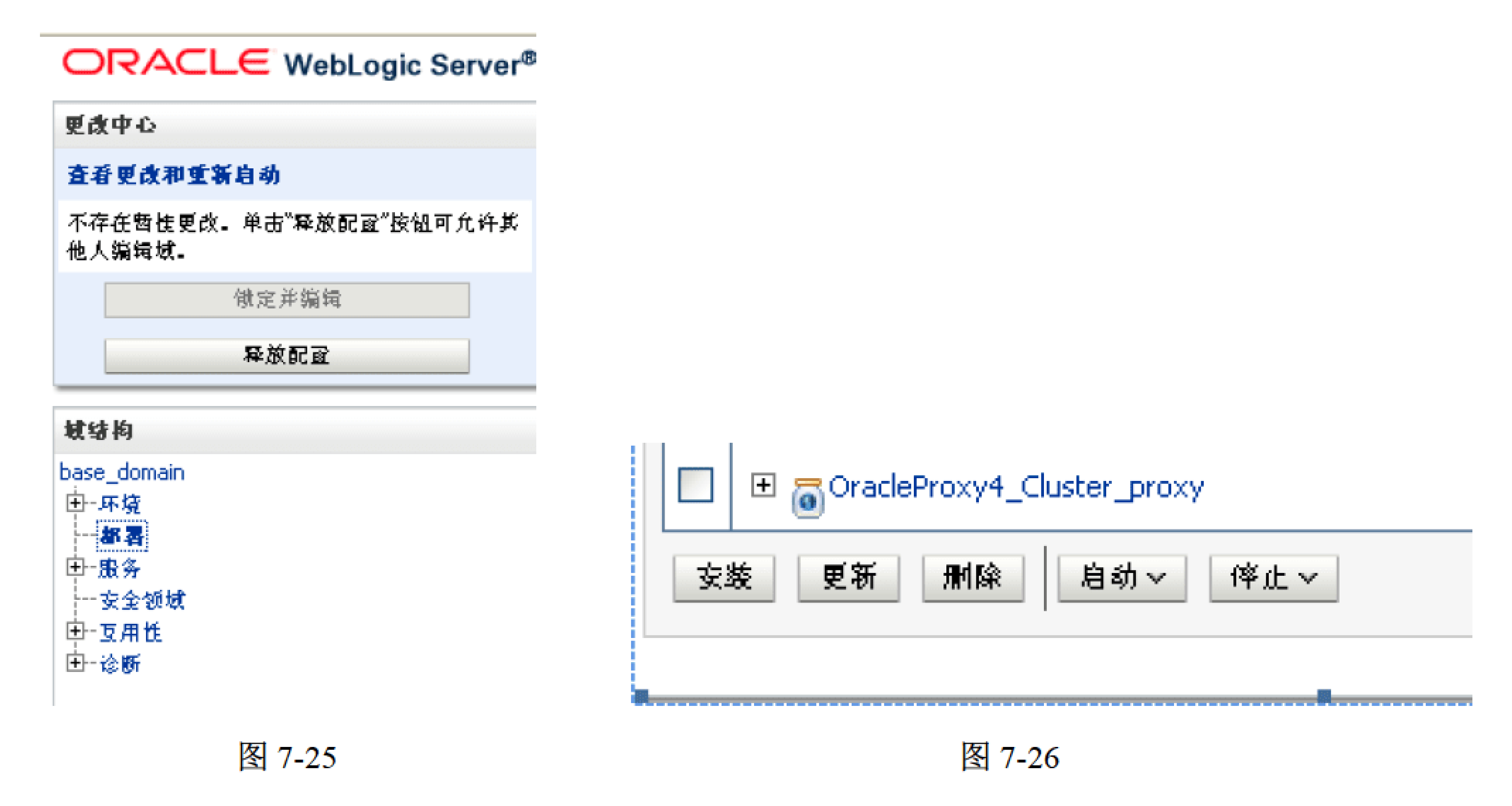


图 7-25　　　　　　　　图 7-26

（3）打开要部署应用的地址，选择要部署的应用，单击“下一步”按钮。

（4）选择定位模式，如果是应用就选择“将此部署安装为应用程序”单选按钮，如果是供给应用调用的公用包，如 EJB，就选择“将此部署安装为库”单选按钮，如图 7-27 所示。



图 7-27

（5）选择部署到哪个服务器，这里我们测试的是集群，所以选中 Cluster 复选框，如图 7-28 所示。

（6）这里是一些部署可选配置，名称为您访问时对应的应用名，安全属性以后会做介绍，这里选择默认选项，因为是集群，记得选择“将此应用程序复制到每个目标”单选按钮，如图 7-29 所示。

上一步 下一步 完成 取消

选择部署目标

选择要对其部署此应用程序的服务器和/或集群。(可稍后重新配置部署目标)。

bs的可用目标：

服务器
☐ AdminServer
☐ proxy

集群
☑ Cluster ◉ 集群中的所有服务器 ○ 集群的一部分 ☐ ms1 ☐ ms2

上一步 下一步 完成 取消

图 7-28

安装应用程序辅助程序

上一步 下一步 完成 取消

可选设置

您可以修改这些设置或接受默认设置

— 一般信息

您希望如何命名此部署？

名称: bs

— 安全

您希望该应用程序使用什么安全模型？

◉ 仅部署描述符: 仅使用在部署描述符中定义的角色和策略。

○ 定制角色: 使用在管理控制台中定义的角色; 使用在部署描述符中定义的策略。

○ 定制角色和策略: 仅使用在管理控制台中定义的角色和策略。

○ 高级: 使用在该领域的配置页上已配置的定制模型。

— 源可访问性

应如何使该源文件可访问？

○ 使用部署的目标定义的默认值

推荐的选择。

◉ 将此应用程序复制到每个目标

在部署过程中，文件将被自动复制到向其定位应用程序的受管服务器上。

○ 我要使部署能够通过下列位置进行访问

位置: E:\bs

提供所有目标将访问此应用程序文件的位置。这通常是一个共享目录。必须确保应用程序文件存在于此位置，且

上一步 下一步 完成 取消

图 7-29

单击“完成”按钮后完成部署，接下来就是测试集群了。

7.7　集群测试

（1）启动应用服务，如图 7-30 所示。



图 7-30

如果启动成功，则会显示如图 7-31 所示的活动状态。



图 7-31

（2）接下来可以通过浏览器访问应用，测试部署应用成功与否。

控制台输出如图 7-32 所示。

```
<2010-11-19 下午12时13分01秒 CST> <Notice> <Log Management> <BEA-170027> <The Server has established connection
<2010-11-19 下午12时13分02秒 CST> <Notice> <Cluster> <BEA-000138> <Listening for announcements from cluster Clus
<2010-11-19 下午12时13分02秒 CST> <Notice> <Cluster> <BEA-000133> <Waiting to synchronize with other running mem
<2010-11-19 下午12时13分32秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to ADMIN>
<2010-11-19 下午12时13分33秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to RESUMING>
<2010-11-19 下午12时13分33秒 CST> <Notice> <Cluster> <BEA-000162> <Starting "async" replication service with rem
<2010-11-19 下午12时13分33秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default" is now listening on 192.168.
<2010-11-19 下午12时13分33秒 CST> <Notice> <WebLogicServer> <BEA-000330> <Started WebLogic Managed Server "ms1"
<2010-11-19 下午12时13分35秒 CST> <Notice> <Cluster> <BEA-000102> <Joining cluster Cluster on 239.192.0.0:7777>
<2010-11-19 下午12时13分36秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to RUNNING>
<2010-11-19 下午12时13分36秒 CST> <Notice> <WebLogicServer> <BEA-000360> <Server started in RUNNING mode>
begin tx
commit tx
begin tx
```

图 7-32

到这里集群的配置、应用部署已经成功了。

7.8 Session 复制

7.8.1 Session 复制的原理

1. HTTP 会话状态复制

WebLogic Server 使用两种方法来跨集群复制 HTTP 会话状态。

（1）内存中复制。使用内存中复制时，WebLogic Server 会将会话状态从一个服务器实例复制到另一个服务器实例。主服务器在客户端首先连接的服务器上创建主会话状态，在集群中的另一个 WebLogic Server 实例上创建次级副本。该副本总是保持最新状态，当主服务器失败时可以使用该副本。

（2）基于 Session 的持久化。WebLogic Server 可以将 Session 持久化到文件或者 JDBC 数据源，从而达到在各个服务器实例之间共享 Session 信息的目的。

2. HTTP 会话复制流程

（1）代理连接过程。当 HTTP 客户端请求 Servlet 时，HttpClusterServlet 将该请求代理传输到 WebLogic Server 集群。HttpClusterServlet 维护集群中所有服务器的列表以及访问集群时要使用的负载平衡逻辑。在上面的示例中，HttpClusterServlet 将客户端请求路由到了 WebLogic Server A 承载的 Servlet。WebLogic Server A 成为了承载该客户端的 Servlet 会话的主服务器。

为了对该 Servlet 提供故障转移服务，主服务器将客户端的 Servlet 会话状态复制到集群中的某个次级 WebLogic Server。这样可确保即使在主服务器失败（例如由于网络失败）时该会话状态的副本仍存在。在上面的示例中，服务器 B 被选择为次级服务器。

Servlet 页通过 HttpClusterServlet 返回到客户端，然后客户端浏览器收到指令，写入列出该 Servlet 会话状态主位置和次级位置的 Cookie。如果客户端浏览器不支持 Cookie，WebLogic Server 则可以使用 URL 重写来代替。

（2）使用 URL 重写跟踪会话副本。WebLogic Server 的默认配置使用客户端 Cookie 来跟踪承载客户端 Servlet 会话状态的主服务器和次级服务器。如果客户端浏览器禁用了 Cookie 的使用，WebLogic Server 还可以使用 URL 重写来跟踪主服务器和次级服务器。使用 URL 重写时，两个位置的客户端会话状态都会嵌入到在客户端和代理服务器之间传递的 URL 中。为了支持此功能，您必须确保在 WebLogic Server 集群上启用了 URL 重写。有关如何启用 URL 重写的说明，可参阅“使用会话和会话持久性”中的使用 URL 重写功能。

（3）代理故障转移过程。如果主服务器失败，HttpClusterServlet 就使用客户端的 Cookie 信息来确定承载会话状态副本的次级 WebLogic Server 的位置。HttpClusterServlet 会自动将客户端的下一个 HTTP 请求重定向到次级服务器，故障转移对于客户端是透明的。

发生失败之后，WebLogic Server B 成为承载 Servlet 会话状态的主服务器，并且会创

建新的次级服务器（在上面示例中为服务器 C）。在 HTTP 响应中，代理会更新客户端的 Cookie 来反映新的主服务器和次级服务器，以考虑后续故障转移的可能性。

在由两个服务器组成的集群中，客户端将以透明方式故障转移到承载次级会话状态的服务器。但是，客户端会话状态的复制不会继续，除非另一个 WebLogic Server 变为可用状态并加入该集群。例如，如果原始主服务器重新启动或重新连接网络，则会使用它来承载次级会话状态。

7.8.2　Session 复制的配置

1. 内存中复制

要配置内存中复制，请执行下列操作。

（1）配置代理服务器（如果适用）。

（2）配置复制组和（或）计算机（可选）。

（3）在 weblogic.xml 部署描述符中指定持久性类型。

配置内存中复制会话持久性。

示例 7-4：

```
<session-descriptor>
<persistent-store-type>replicated</persistent-store-type>
</session-descriptor>
```

可选参数如下。

① memory——禁用持久性会话存储。

② replicated——与 memory 相同，但会话数据将在集群服务器之间复制。

③ replicated_if_clustered——如果 Web 应用程序部署在集群服务器上，则按有效的 persistentstore-type 复制；否则按默认值 memory 复制。

④ sync-replication-across-cluster——复制将在集群内同步发生。

⑤ async-replication-across-cluster——复制将在集群内异步发生。

2. 配置 JDBC 复制

要配置 JDBC 复制，可执行下列操作。

（1）在数据库中创建表。

（2）创建对数据库具有读/写权限的连接池。

（3）在 weblogic.xml 部署描述符中配置会话持久性。

配置会话持久性的示例如下。

示例 7-5：

```
<session-descriptor>
<persistent-store-type>jdbc</persistent-store-type>
<persistent-store-pool>SessionDS</persistent-store-pool>
<persistent-store-table>WL_SERVLET_SESSIONS</persistentstore-
```

```
table>
</session-descriptor>
```

JDBC 持久性表配置必须存在名为 WL_SERVLET_SESSIONS 且具有读/写访问权限的数据库表，见表 7-2。

表 7-2

列标题	列数据类型
WL_ID	可变宽度，最多 100 个字符
WL_CONTEXT_PATH	
WL_IS_NEW	numeric，20 位
WL_ACCESS_TIME	numeric，20 位
WL_SESSION_VALUES	BLOB，非常大
WL_IS_VALID	char，1 个字符
WL_CREATE_TIME	numeric，20 位

在映射到会话持久性连接池的数据库中，需要配置一个名为 WL_SERVLET_SESSIONS 的表，该表将包含所有活动会话对象的值。指定有权访问此表的用户，需要对该表具有读/写/插入/删除访问权限才能有效地管理对象。该表需具有以下 8 列内容。

- **WL_ID**——会话 ID，与 WL_CONTEXT_PATH 一起用做数据库主键。可变宽度字母数字数据类型，最多 100 个字符。
- **WL_CONTEXT_PATH**——上下文，此列与 WL_ID 一起用做主键。可变宽度字母数字数据类型，最多 100 个字符。
- **WL_IS_NEW**——只要 Servlet 引擎将会话分类为“新”状态，此值就为 True。含一个字符的列。
- **WL_CREATE_TIME**——最初创建会话的时间。Numeric 数据类型列，20 位。
- **WL_IS_VALID**——Servlet 可以访问会话时，该值为 True。用于并发访问。含一个字符的列。
- **WL_SESSION_VALUES**——实际会话数据。BLOB 列。
- **WL_ACCESS_TIME**——上次访问会话的时间。Numeric 数据类型列，20 位。
- **WL_MAX_INACTIVE_INTERVAL**——从客户端请求到会话失效之间的时间(秒)。负时间值表明会话永不超时。Integer 数据类型列。

创建此表的 SQL 语句示例如下（适用于 Oracle）。

示例 7-6：

```
create table wl_servlet_sessions ( wl_id VARCHAR2(100) NOT NULL,
wl_context_path VARCHAR2(100) NOT NULL, wl_is_new CHAR(1),
wl_create_time NUMBER(20), wl_is_valid CHAR(1), wl_session_values LONG
RAW, wl_access_time NUMBER(20), wl_max_inactive_interval INTEGER,
PRIMARY KEY (wl_id, wl_context_path) );
```

3．配置文件持久性

要为 Web 应用程序配置基于文件的会话持久性，可执行下列操作。

（1）创建一个集群上的所有服务器都可以共享访问的文件夹。
（2）在 weblogic.xml 部署描述符中指定 file 持久性类型。
配置会话持久性的示例如下。
示例 7-7：

```
<session-descriptor>
<persistent-store-type>file</persistent-store-type>
<persistent-store-dir>shared folder location</persistentstore-dir>
</session-descriptor>
```

7.9　新建启动脚本

启动脚本都是以 Domain 域安装时默认的路径为准，如果您修改了您的 Domain 域的安装路径，可以实际为准。

7.9.1　启动服务器脚本

示例 7-8：

```
startAs.sh:

cd %weblogic_home%/user_projects/domains/domainname/bin
nohup ./startWebLogic.sh &
```



Domainname 是您的域名。

7.9.2　启动被管服务器脚本

示例 7-9：

```
startMs1.sh:

cd %weblogic_home%/user_projects/domains/domainname/bin
nohup ./startManagedWebLogic.sh ms1 http://192.168.0.139:7001 &

startMs2.sh:
cd %weblogic_home%/user_projects/domains/domainname/bin
nohup ./startManagedWebLogic.sh ms2 http://192.168.0.139:7001 &
```

7.9.3 启动代理服务器脚本

示例 7-10：

```
startPs.sh:

cd %weblogic_home%/user_projects/domains/domainname/bin
nohup ./startManagedWebLogic.sh ps http://192.168.0.139:7001 &
```

第 8 章　WebLogic 常用的管理操作

BEA WebLogic Server 包含了许多互相关联的资源。对这些资源的管理包括服务器的启动及终止，服务器以及连接池的负载平衡，资源配置的监控、诊断并修改问题，监控并评估系统性能，分发 Web 应用、EJB 以及其他资源。WebLogic 服务器提供了一个健壮易用的基于 Web 的工具——管理控制台，它是执行上述任务的主要工具。通过管理控制台，可以访问 WebLogic 管理服务。

管理控制台是一个 Web 应用，它使用 JSP 来访问管理服务器所管理的资源。管理服务器启动以后，在浏览器中使用以下 URL 启动管理控制台。http://hostname:port/console，输入用户名跟密码，就可以进入控制台进行常用的管理了。下面我们来一起讨论常用的管理操作。本章示例均在 Windows XP 下运行。

8.1　添加删除服务 Service

使用管理控制台可以配置下列服务。首先介绍一下各服务的相关概念，其中 JDBC 等服务是最常用的。

8.1.1　消息传送

WebLogic JMS 是一种企业级的消息传送系统，完全支持 JMS 规范，还可提供很多超出标准 JMS API 的扩展。它紧密集成在 WebLogic Server 平台中，从而使您可以生成高度安全的 J2EE 应用程序，可通过 WebLogic Server 控制台轻松地对其进行监视和管理。除了完全支持 XA 事务处理，通过 WebLogic JMS 的集群和服务迁移功能也可以得到高可用性，同时还具有与 WebLogic Server 和第三方消息传送供应商的其他版本的无缝互操作性。

8.1.2　JDBC

通过 WebLogic JDBC 服务，您可以在 WebLogic 域中通过数据源和多数据源配置数据库连接。数据源提供数据库连接池和连接管理。多数据源提供数据源之间的负载平衡和故障转移，它可以连接不同的后端资源。

8.1.3　持久性存储

持久性存储是用于存储子系统数据（例如持久性 JMS 消息）的物理资料档案库。它既可以是可通过 JDBC 访问的数据库，也可以是基于磁盘的文件。

8.1.4 路径服务

路径服务是用于存储一组消息和一个消息传送资源（如分布式目标成员或存储转发发送代理）之间的路由路径的工具。

8.1.5 外部 JNDI 提供程序

外部 JNDI 提供程序表示驻留在 WebLogic Server 环境外的 JNDI 树。这可能是不同服务器环境或外部 Java 程序中的 JNDI 树。通过设置外部 JNDI 提供程序，可以像使用 WebLogic Server 实例绑定的对象那样，轻松查找和使用远程对象。

8.1.6 工作上下文

工作上下文允许开发者定义隐式流经远程请求的属性，允许下游组件在调用客户机的上下文中工作。

8.1.7 XML 注册表

XML 注册表是用于配置和管理 WebLogic Server 实例的 XML 资源的工具。WebLogic Server 中的 XML 资源包括应用程序用于对 XML 数据进行语法分析的语法分析器，应用程序用于转换 XML 数据的转换器，外部实体解析和外部实体高速缓存。

8.1.8 XML 实体高速缓存

XML 实体高速缓存存储通过 URL 或与 EAR 档案主目录相对的路径名所引用的外部实体。高速缓存外部实体可节省远程访问时间，以及当对 XML 文档进行语法分析时，在无法访问管理服务器的情况下提供本地备份。

8.1.9 jCOM

WebLogic jCOM 是软件桥接程序，用其可在 WebLogic Server 中部署的 Java/J2EE 对象与 Microsoft Office 系列产品中的 Microsoft ActiveX 组件、Visual Basic 和 C++对象，以及其他组件对象模型/分布式组件对象模型（COM/DCOM）环境之间进行双向访问。

8.1.10 邮件会话

WebLogic Server 中包含了 JavaMail API 1.1.3 引用实现。通过 JavaMail API，您可以将电子邮件功能添加到 WebLogic Server 应用程序中。JavaMail 可以使 Java 应用程序访问您网络中或 Internet 上具备 POP3/IMAP 和 SMTP 功能的邮件服务器。

8.1.11　File T3

通过 WebLogic File（T3）服务（已废弃），您可以从客户端高速访问服务器上的本地操作系统文件。使用客户机 API 可以扩展 java.io.InputStream 和 java.io.OutputStream 的功能。

8.1.12　JTA

WebLogic Server 的一个最重要的基本功能是事务处理管理。事务处理是确保正确完成数据库更改的方法，并且保证它们具有高性能事务处理的所有 ACID（原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability））属性。

学习了这些概念后，我们就可以用控制台向导较容易地添加与删除各种服务了。

8.2　Machine 配置

计算机是承载一个或多个 WebLogic Server 实例（服务器）的计算机的逻辑表示。WebLogic Server 使用配置的计算机名来确定集群中某些特定任务（如 HTTP 会话复制）要委派到的最佳服务器。管理服务器使用此计算机定义和节点管理器一起来启动远程服务器。

通过单击 Machines 查看域中有多少台计算机，其中可以从图 8-1 中看到 Node Manager 的状态。

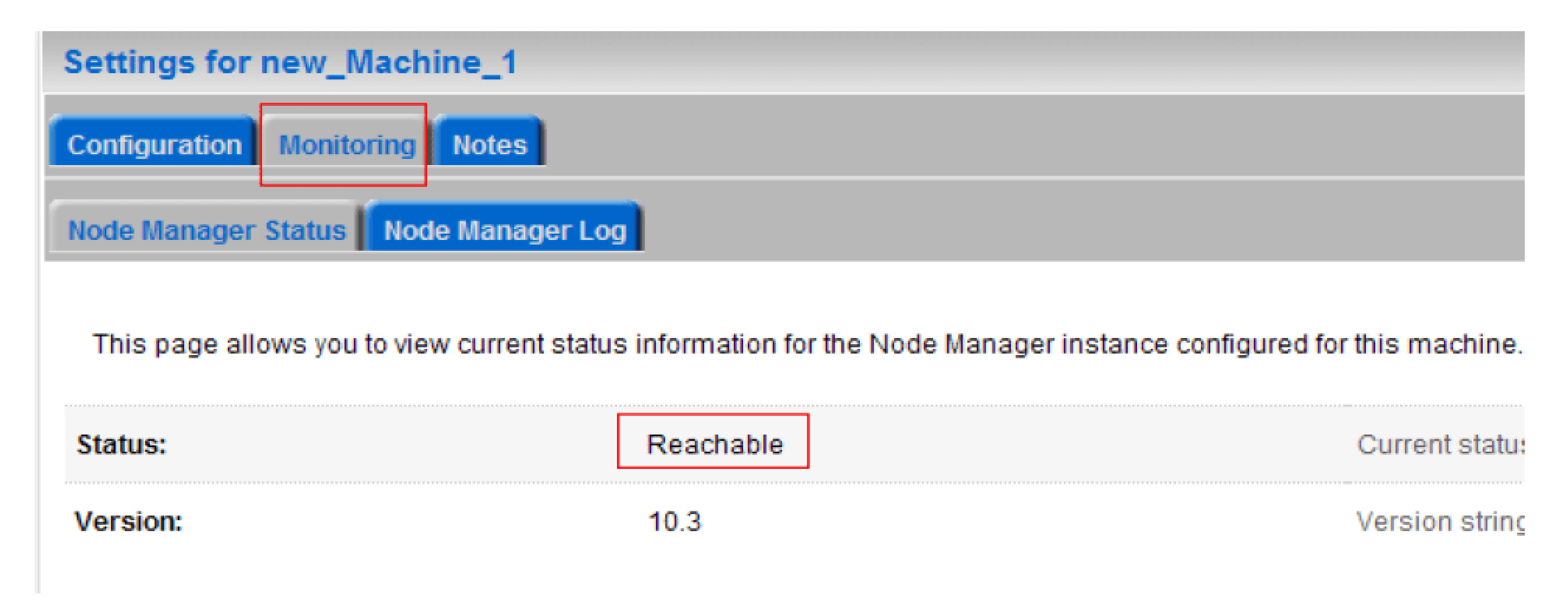


图 8-1

这儿可以看到 Status 为 Reachable，说明 Node Manager 处于活动状态。

8.3　JDBC 配置

下边来演示新建删除一个 JDBC 服务。

首先选择 Service 里的 JDBC 下的 Data Sources 选项，如图 8-2 所示。

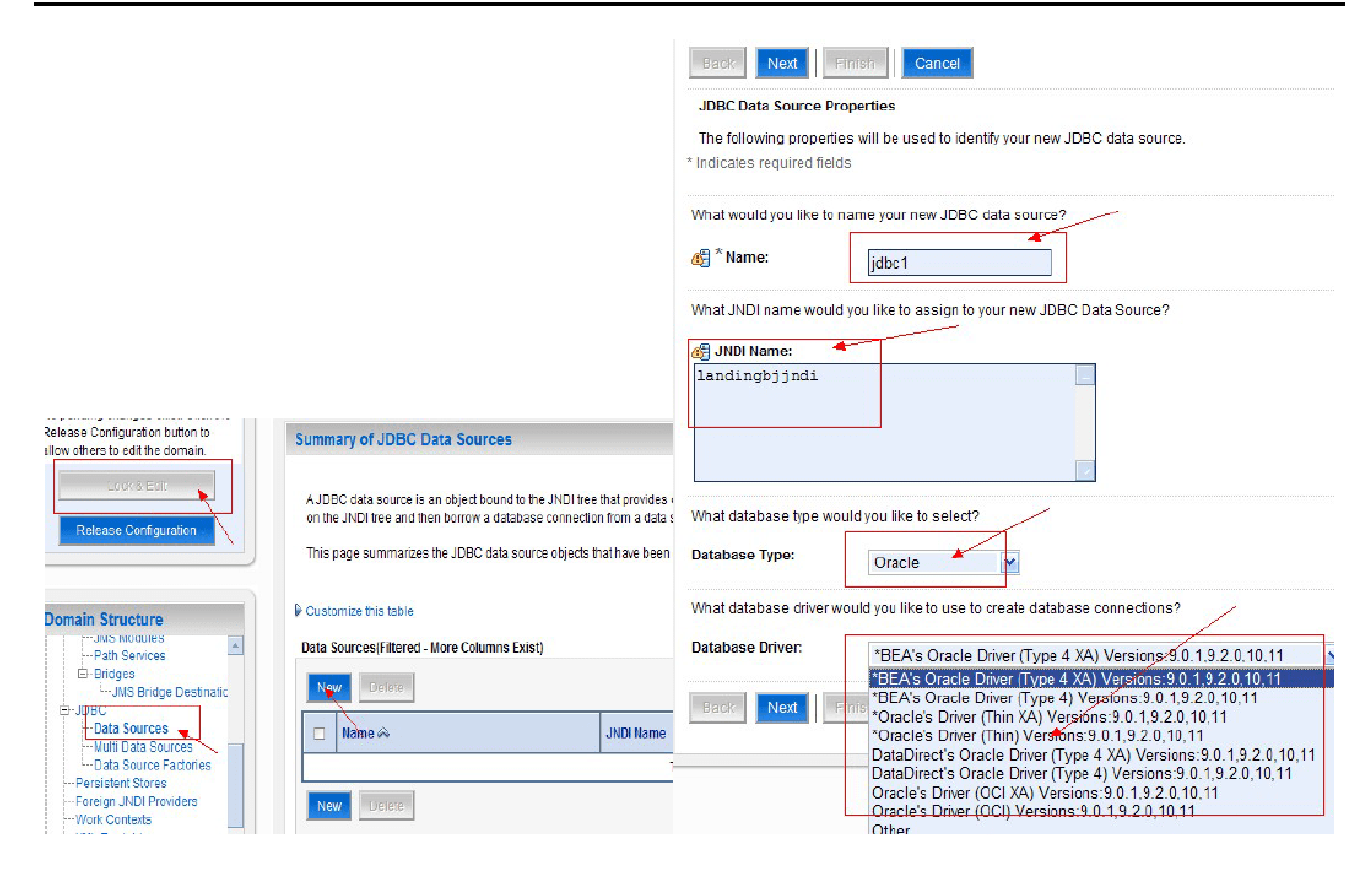


图 8-2

再选择 Lock&Edit 选项，新建一个 JDBC 源，Name 是任意起的，在这起名叫做 jdbc1，JNDI Name 是应用中用来引用该数据源名字的，图片中为 landingbj jndi。

再选择后台数据库的类型，这里假设为 Oracle，接着选择 Database Driver 区域中提供的 Oracle 自己的驱动与 BEA 提供的驱动，其中带 XA 字样的驱动为 WebLogic 支持事务处理。

接着单击 Next 按钮，如图 8-3 所示。接着就提示要输入数据库相关的参数了。我们可以先去 Oracle 数据库看看相关参数名，用 show parameter name 可以看到 db_name 为 CHENG，如图 8-4 所示。所以接下来的方框里内容如图 8-5 所示。

Back　Next　Finish　Cancel

Transaction Options

You have selected an XA JDBC driver to use to cr
Phase Commit' global transaction protocol. No o

Back　Next　Finish　Cancel

图 8-3

```
SQL> show parameter name

NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------
db_file_name_convert                 string
db_name                              string      CHENG
db_unique_name                       string      CHENG
global_names                         boolean     FALSE
instance_name                        string      cheng
lock_name_space                      string
log_file_name_convert                string
service_names                        string      CHENG
sp_name                              string      CHENG
SQL>
```

图 8-4

此处选择 Oracle 数据库的默认 scott 用户做测试，如图 8-6 所示。

图 8-5

图 8-6

单击 Next 按钮，就可以看到向导根据我们的配置自动生成的 Driver Class Name 和 URL，再次单击 Test Configuration 按钮，把数据库的监听打开，一般情况下就可以看到 connection test succeed 了。

接下来就可以把 JDBC 服务部署到集群中了。

单击 Targets 按钮，如图 8-7 所示。

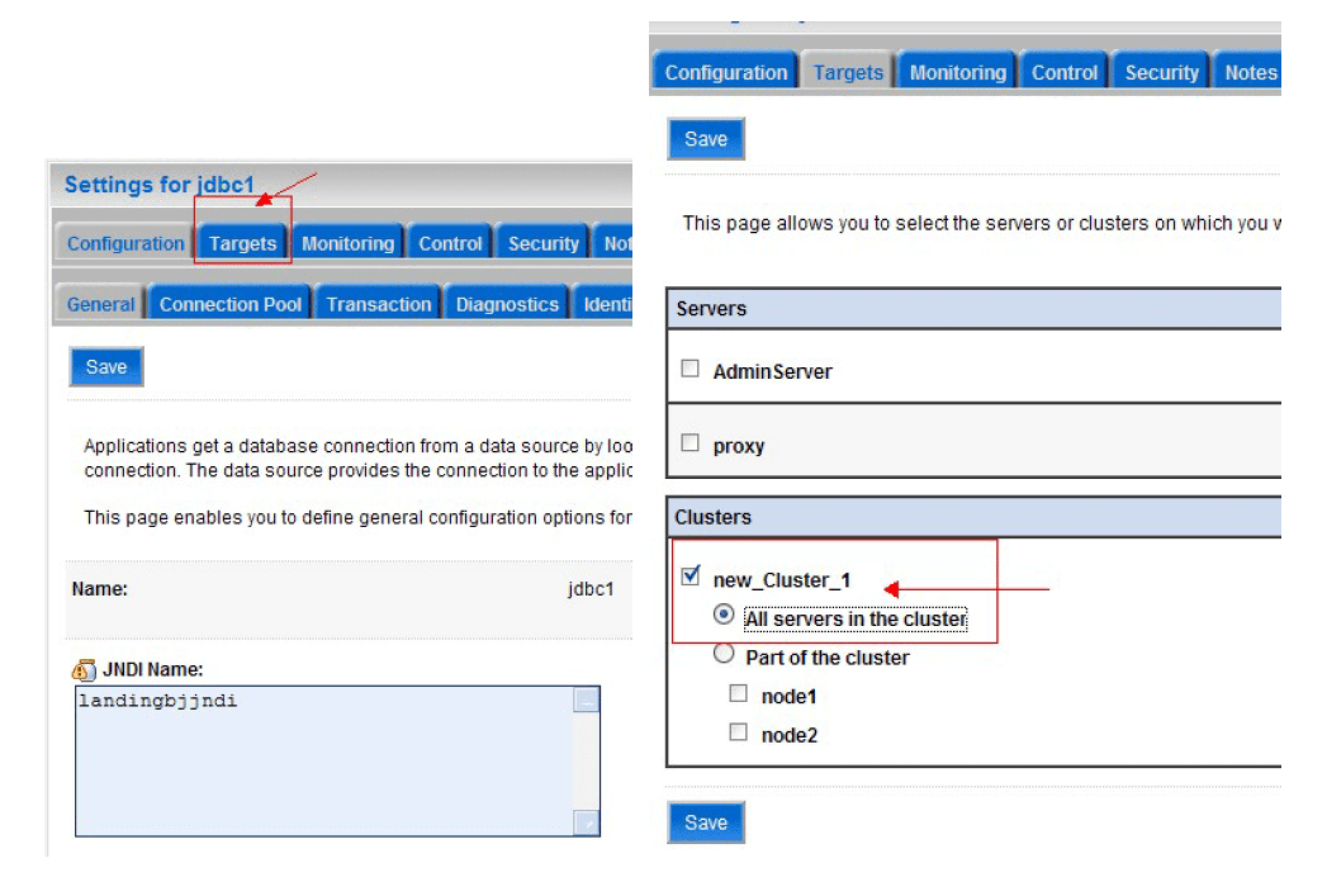


图 8-7

单击 Next 按钮，然后激活，就可以看到激活成功字样，这样就完整地添加了一个 JDBC 源，且把这个服务部署到了集群中，如图 8-8 和图 8-9 所示。

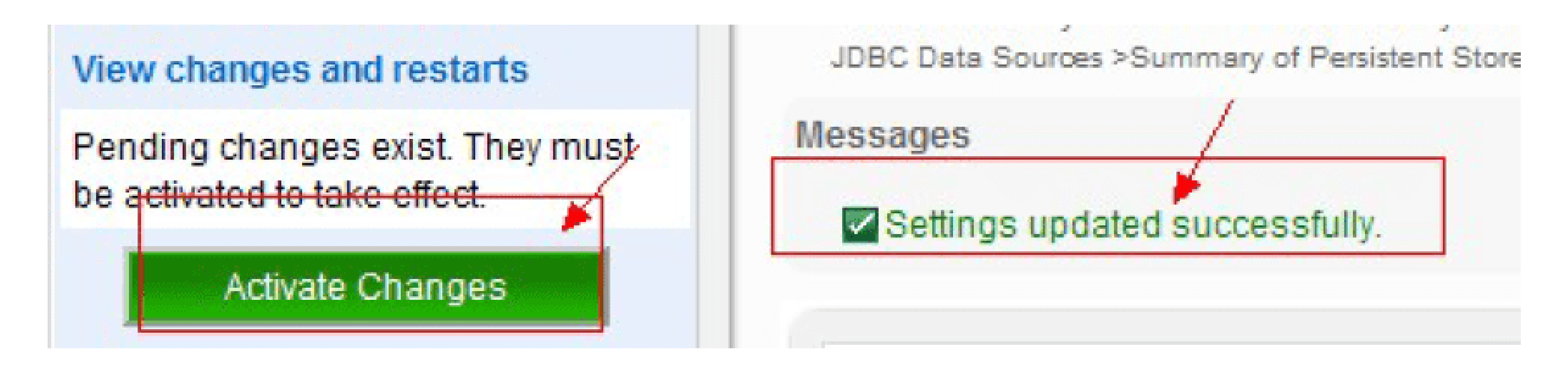


图 8-8

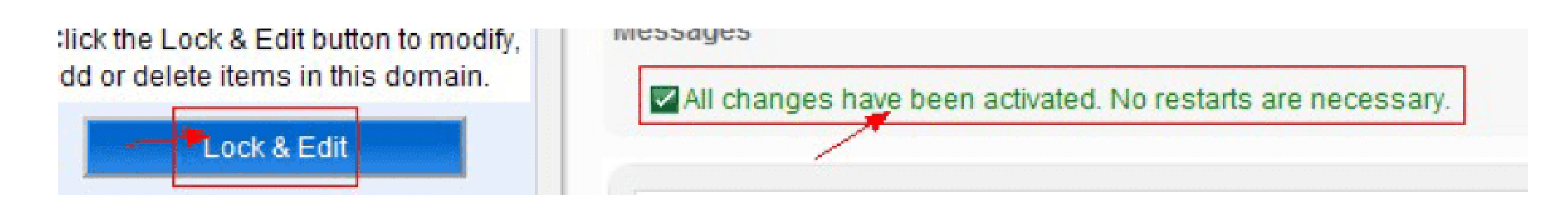


图 8-9

有的时候，我们需要将已有的数据源删除，下边一起来演示一下。

首先要确定 JDBC 没有被其他服务调用，有的话删掉就可以，且 JDBC 源从集群中剔除，然后编辑、删除就可以了，如图 8-10 所示。

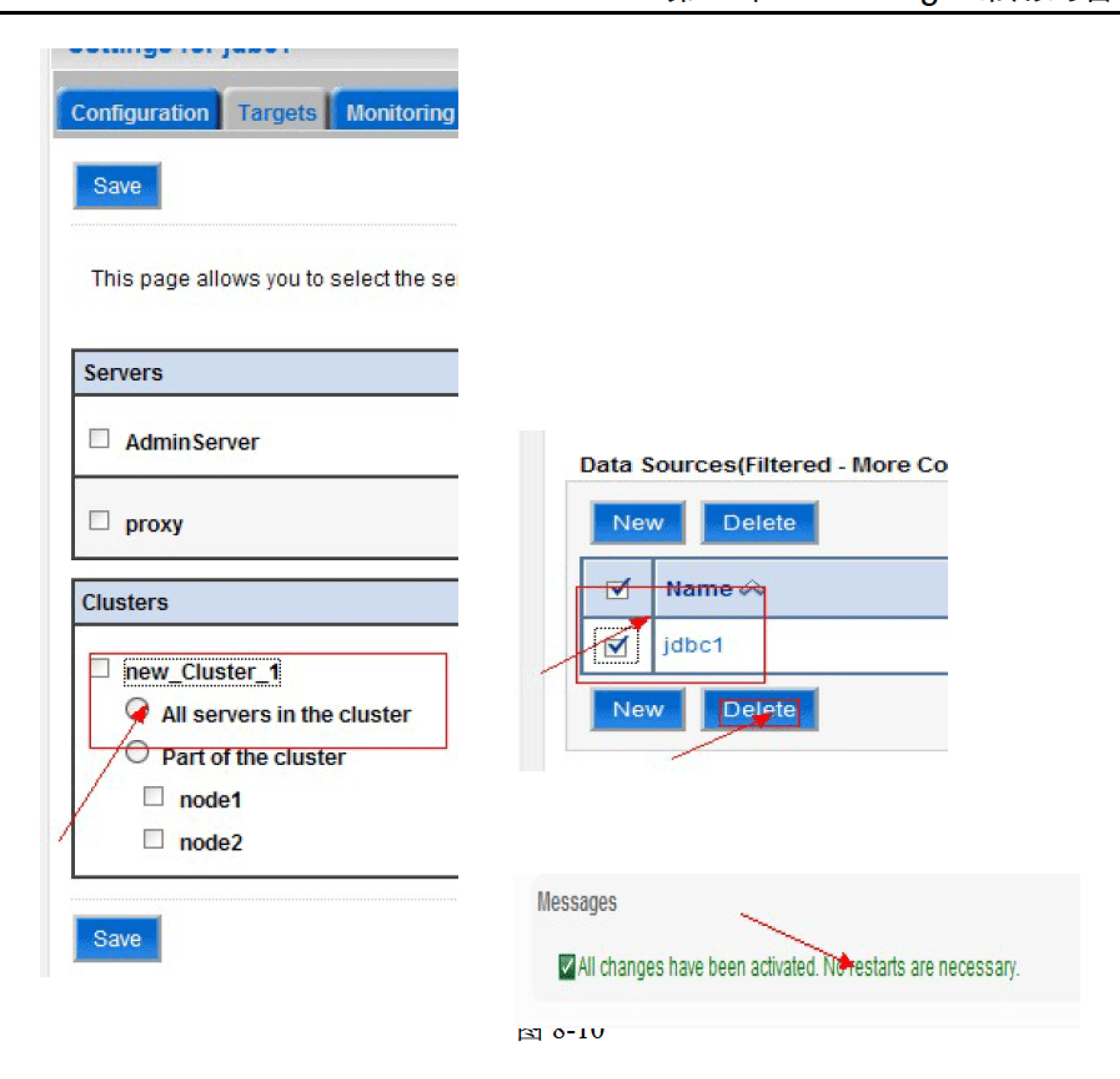


图 8-10

8.4 Node Manager 的配置

WebLogic Server 生产环境中的服务器实例通常分布在多个域、计算机和地理位置上。节点管理器是一个 WebLogic Server 的实用工具，可以使用它来启动、关闭和重新启动远程位置上的管理服务器和受管服务器实例。尽管节点服务器是可选工具，但如果您的 WebLogic Server 环境所承载的应用程序具有较高的可用性要求，则建议您使用该工具。

节点管理器进程与特定 WebLogic 域无关，而与计算机有关。您可以使用同一个节点管理器进程控制任意 WebLogic Server 域中的服务器实例，只要该服务器实例与节点管理器进程驻留于同一台计算机上。节点管理器必须在承载要通过节点管理器控制 WebLogic Server 实例（管理服务器或受管服务器）的计算机。

WebLogic Server 提供两种版本的节点管理器：基于 Java 的节点管理器和基于脚本的节点管理器，两者功能类似。但是，每个版本的配置和安全注意事项不同。

下边以图 8-11 来简单描述一下节点管理器在整个域的大致关系。

在 Windows 下 Node Manager 的配置较简单，安装的时候选择 Yes 单选按钮，然后启动的时候执行“程序”→WebLogic Server→Tools→Node Manager 命令就可以正常启动了。启动之后可以通过控制台管理受管服务器的启停，如图 8-12 所示。

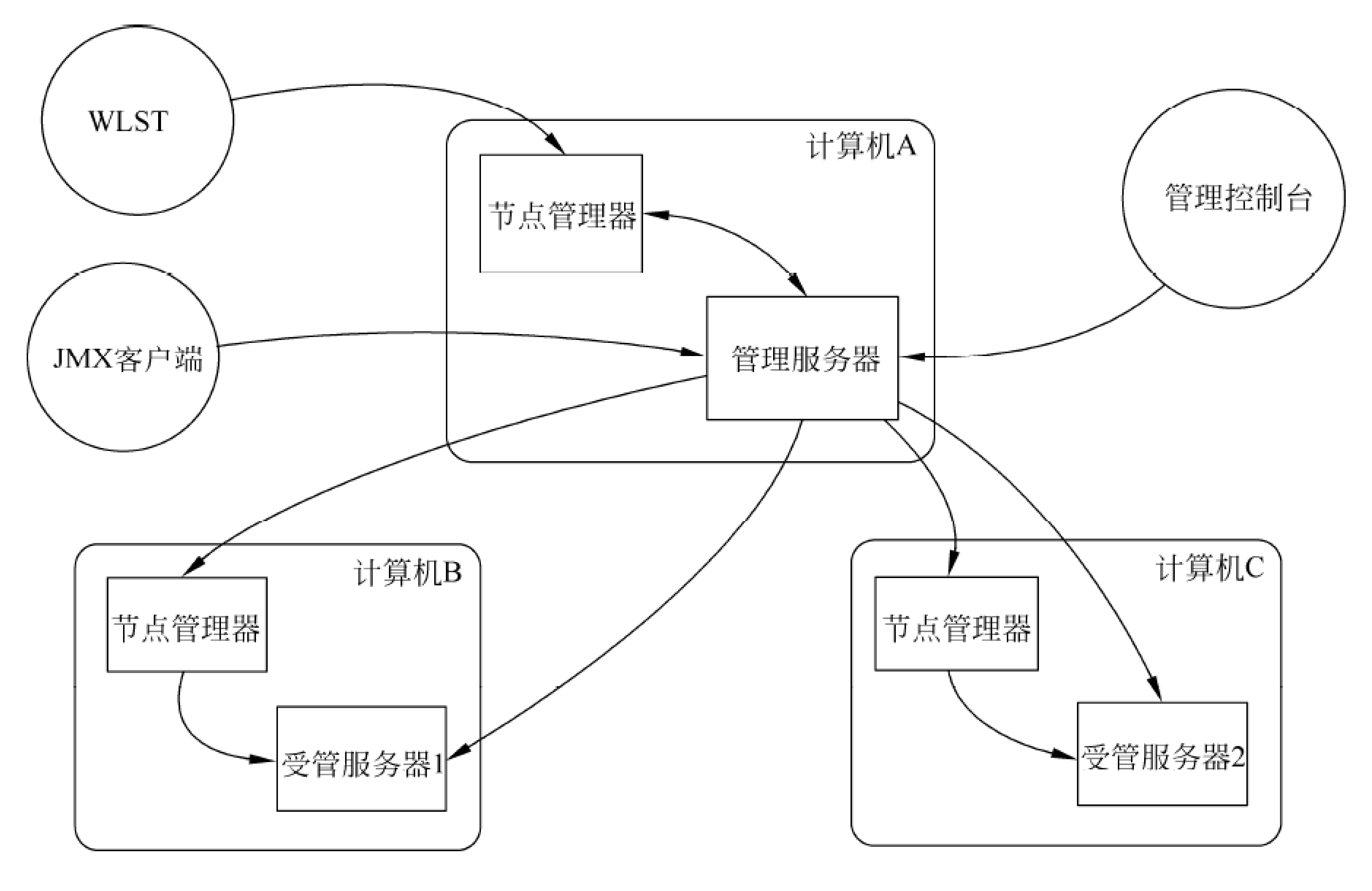


图 8-11

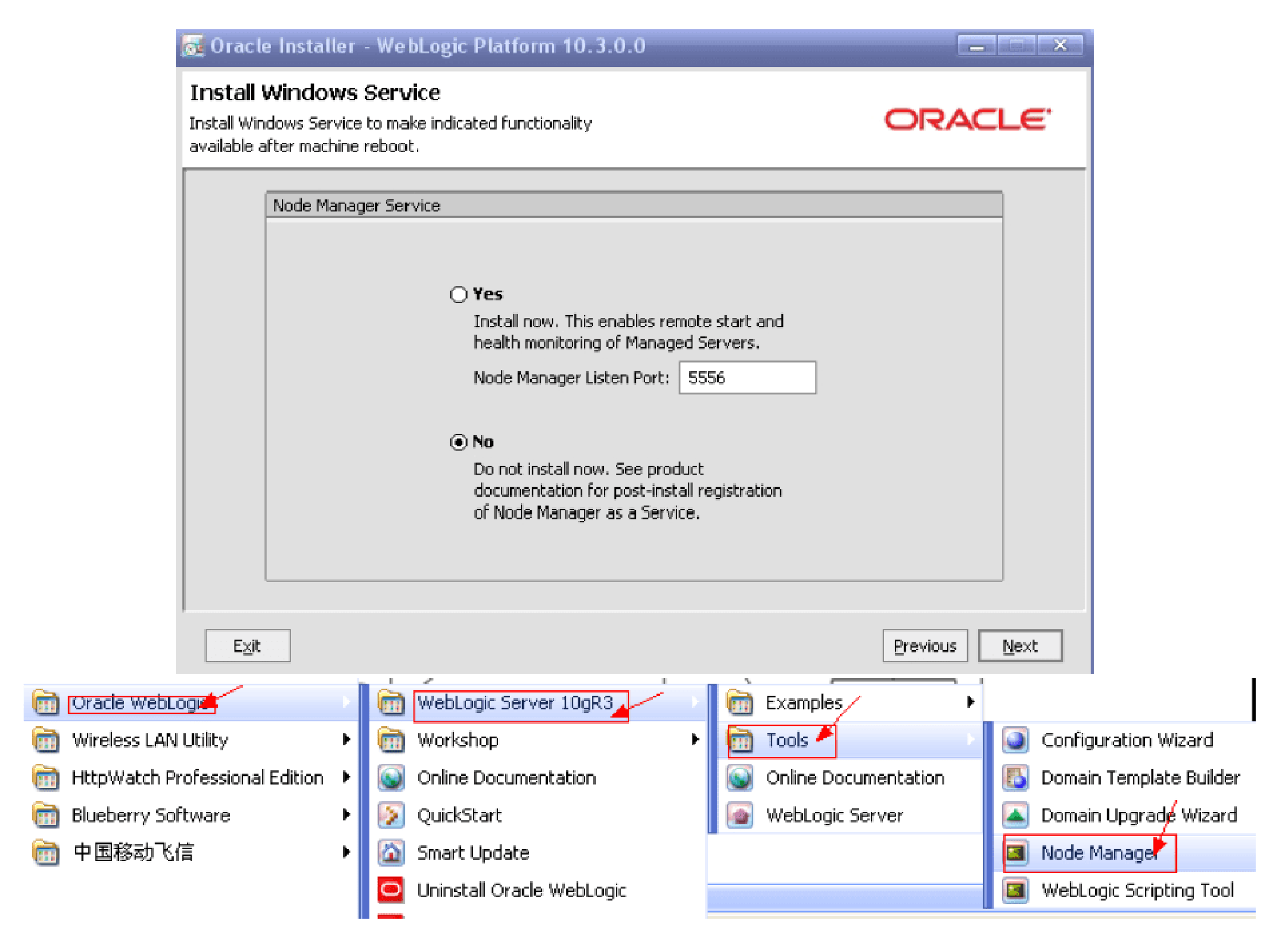


图 8-12

8.5 JMS 配置

JMS 服务器是与环境相关的配置实体，可用做定位到该服务器的 JMS 模块中队列和

主题的管理容器。对于其目标，JMS 服务器的主要用途是维护有关所有到达目标的持久性消息所使用的持久性存储的信息，以及维护在目标上创建的持久订阅者的状态。JMS 服务器还可以管理目标上的消息分页，并且根据需要，还可以为其已定位的目标管理消息或字节阈值以及服务器级别的配额。作为已定位的目标的容器，对 JMS 服务器所做的任何配置或运行更改均会影响其所有目标。

配置 JMS 服务器，可执行下列操作。

（1）在控制台新建一个 JMS 服务器，配置名称、持久性存储，并选择把它部署到需要的目标上。

（2）如图 8-13 所示的操作，选择 JMS Servers 选项，新建一个 JMS 服务器，并起名为 jms。选择配置存储的位置。Paging Directory 指定当 JMS 服务器中消息正文的大小超过消息缓冲区大小时将消息正文写入的位置。其他一些参数可参照旁边的解释依次配置就行了，如图 8-14 所示。

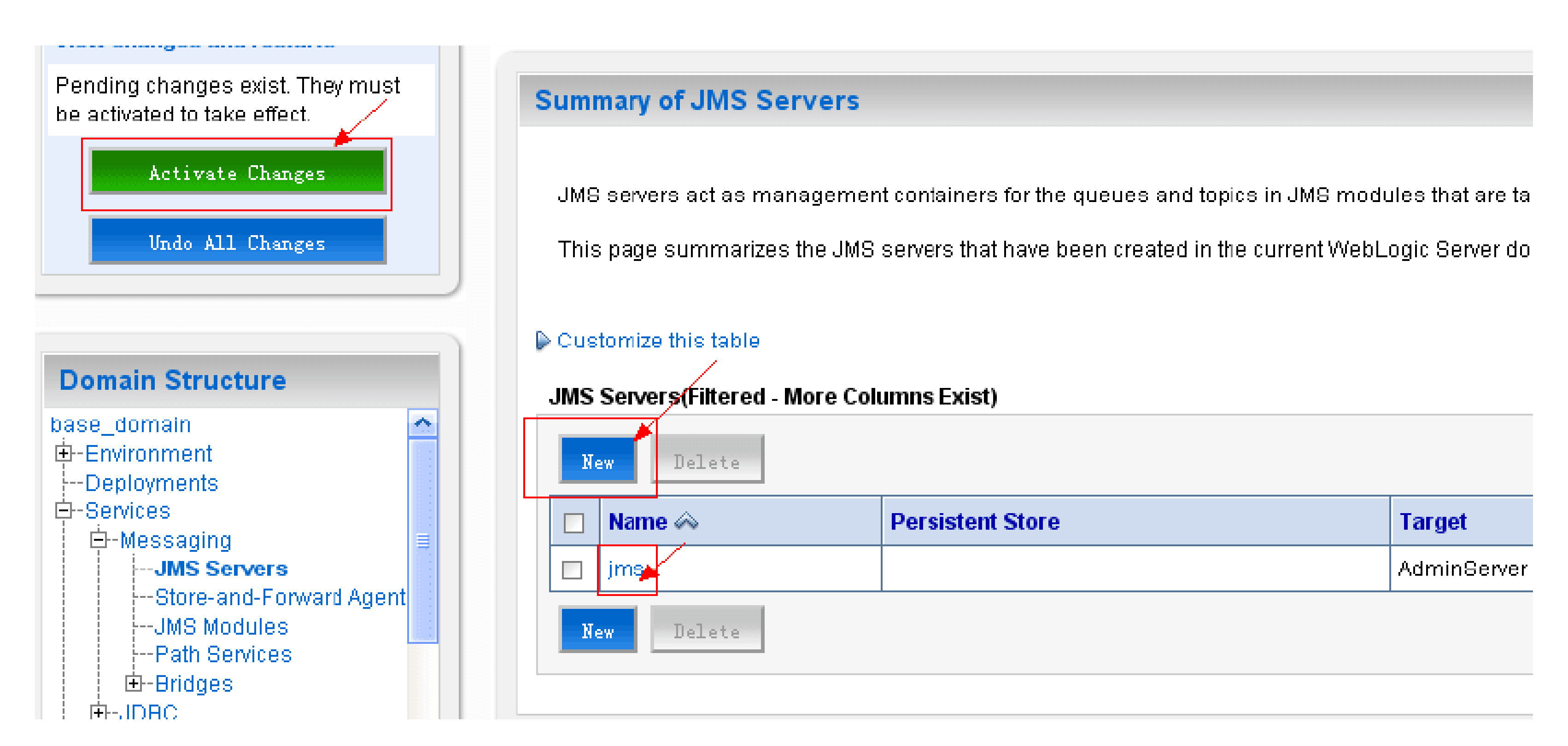


图 8-13

Settings for jms

Configuration | Logging | Targets | Monitoring | Control | Notes

General | Thresholds and Quotas | Session Pools

Save

JMS servers act as management containers for the queues and topics in JMS modules that are destinations is to maintain information on what persistent store is used for any persistent mess durable subscribers created on the destinations.

Use this page to define the general configuration parameters for this JMS server.

Name: jms

Persistent Store: (none)

Paging Directory:

Message Buffer Size: -1

图 8-14

8.6　WTC 配置

WTC 为 WebLogic Tuxedo Connector 的简称，WebLogic Server 客户机可以通过 Tuxedo 服务和 Tuxedo 客户机来调用 WebLogic Server Enterprise Java Bean（EJB）以响应服务请求。WTC 服务器可以启用应用程序、Tuxedo 和 Tuxedo 客户机之间的通信。要配置 WTC 服务，可执行下列操作。

（1）如果尚未执行此操作，可在管理控制台的更改中心中单击“锁定并编辑”按钮。

（2）在管理控制台中，展开“互操作性”选项，然后选择“WTC 服务”选项。

（3）在“WTC 服务器”页上，单击 WTC 服务的名称。

（4）选择“定位和部署”选项卡。

（5）要将 WTC 服务分配到选定的服务器，可从独立服务器列表中选中该服务器，也可以将一个 WTC 服务分配到一台服务器中。

（6）要从服务器中删除 WTC 服务，可从服务器清除选中标记。

（7）单击“保存”按钮。

（8）要激活这些更改，可在管理控制台的更改中心中单击“激活更改”按钮。

配置 Local Tuxedo Access Point 的步骤大致如下。

（1）单击图 8-15 中的 Local APs 按钮。

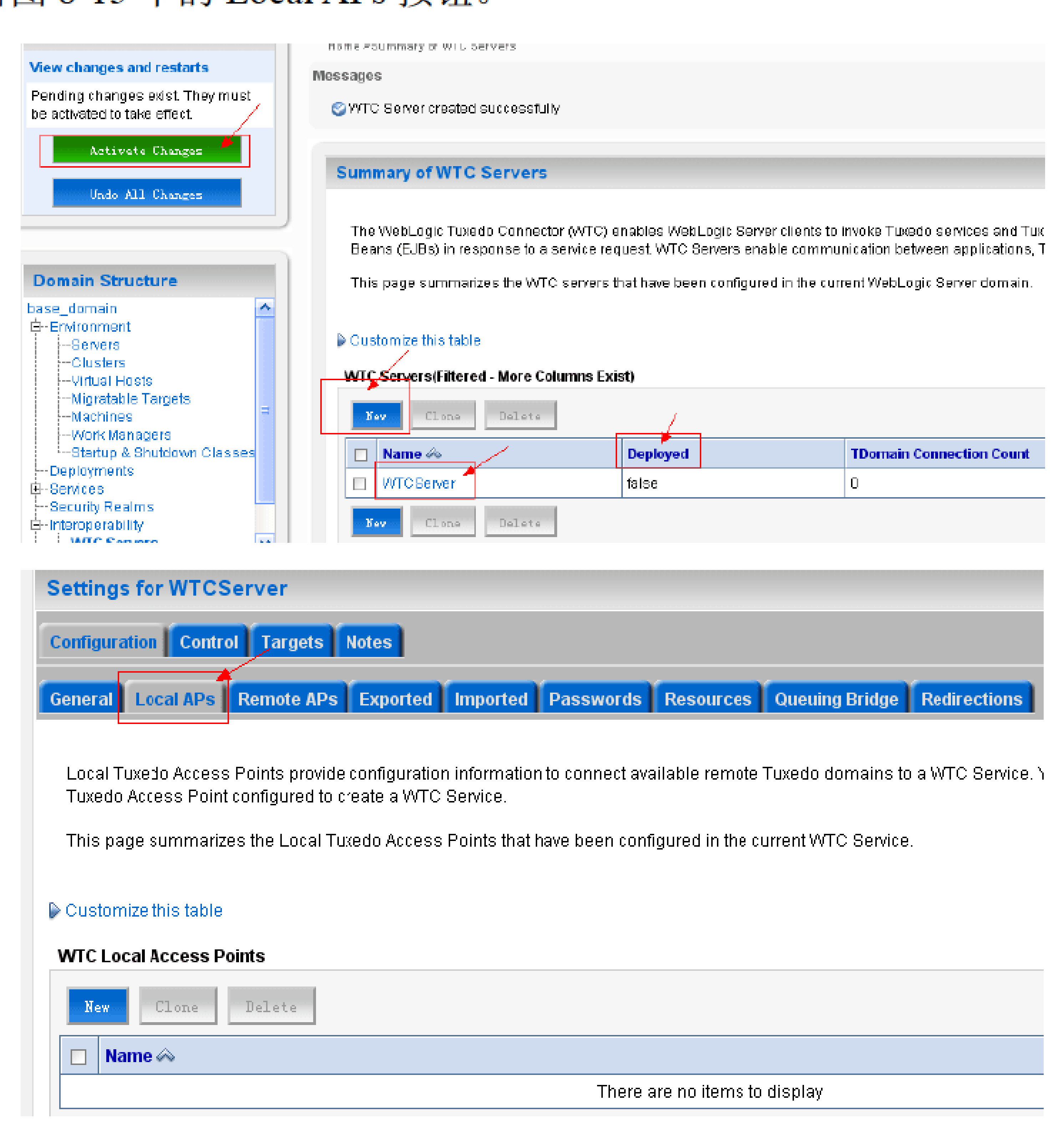


图 8-15

（2）输入如下值。

Access Point:myLocalAp。

Access Point ID:TDOM2。

Network address 此本地 Tuxedo 访问点的网络地址和端口号。格式为 TCP/IP 地址，即 //hostname:port_number 或 //#.#.#.#:port_number。

Example://123.123.123.123:5678。

（3）单击 OK 按钮。

配置远程 Tuxedo Access Point 步骤如下。

（1）单击 Remote APs tab 页。

（2）在该页面可以输入如下值（范例）。

Access point:myRemoteAP。

AccessPoint ID:TDOM1。

Local Access Point:myLocalAp。

Network address ://123.123.123.123.1234（这个 IP 应根据您的真实环境值去写）。

（3）单击 OK 按钮。

创建已导出、导入。步骤大致为单击 Exported tab 页，再输入下面代码：

Resource name:TOLOWER

Local Access Point myLocalApp

EJB name:tuxedo.services.TOLOWERHome

Remote name:TOLOWER

然后单击 OK 按钮。导入的步骤同导出。

再将 WTC 部署到服务器中，选中安全领域的 myrealm 复选框，如 8-16 所示。

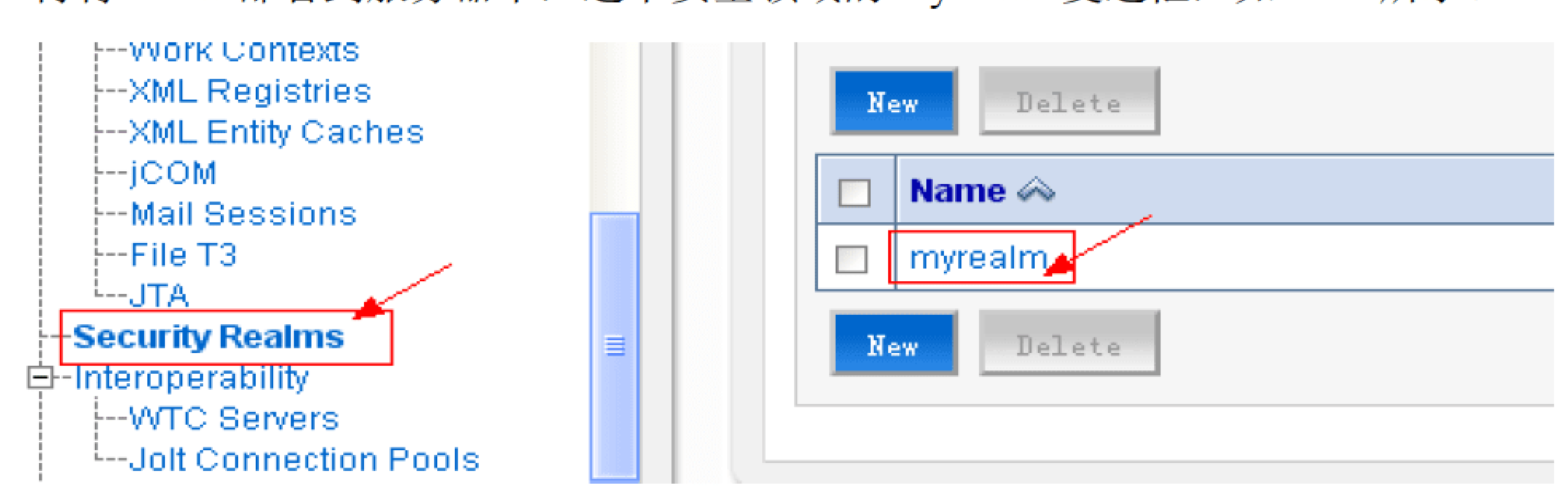


图 8-16

最后单击用户和组，新建一个名为 TDOM1 的用户，锁定并编辑该用户。

8.7　内存参数的修改

通过查看 WebLogic 的启动脚本，就可以很快地知道上哪去修改内存参数了。其中 startManagedWebLogic.cmd 里头有这么一行 call "%DOMAIN_HOME%\bin\setDomainEnv.

cmd" %*。

这儿就知道了去 setDomainEnv.cmd 找设置内存参数的相关行。set MEM_ARGS=-Xms256m -Xmx512m 找到这一行，更改为自己需要的值就可以了，如图 8-17 所示。

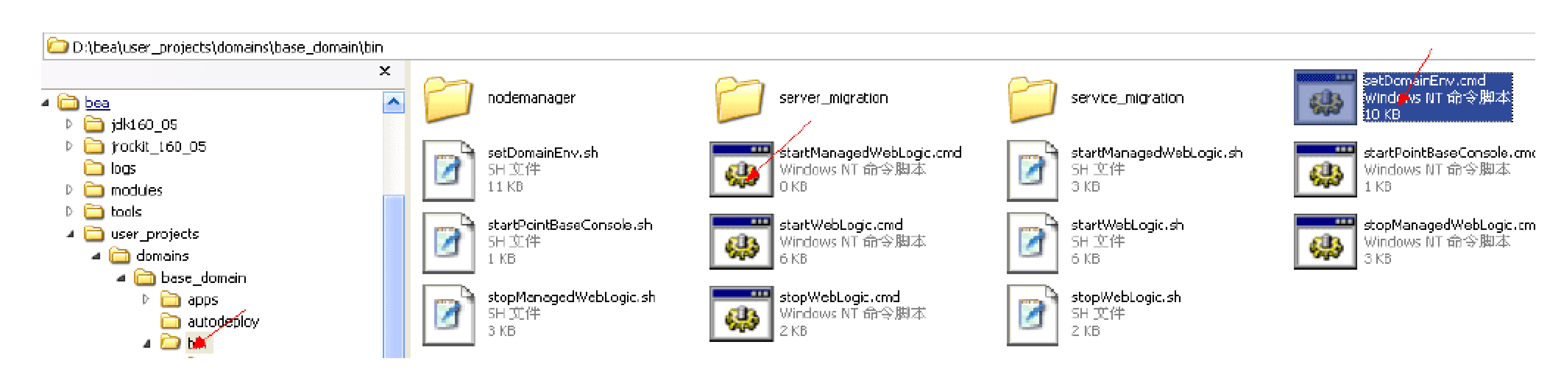


图 8-17

其中参数 Xms 表示启动时 Java 堆栈内存的最小值，Xmx 为相应的最大值。

8.8 更换 JDK

创建域时，如果选择自定义配置，则 Configuration Wizard 会显示 WebLogic Server 安装的 SDK 列表。在此列表中，您可以选择希望运行域的 JVM，然后 Configuration Wizard 将根据您的选择配置 WebLogic 启动脚本。

创建域后，如果希望使用其他 JVM，则可以修改脚本（setDomainEnv.cmd），如下所示。

（1）更改 JAVA_HOME 变量的值。

指定要使用的 SDK 的顶级目录的绝对路径名。例如，c:\bea\jrockit90。

在 Windows 或 Linux 平台上，BEA Systems 建议使用下列 JVM。

① 对于开发模式，使用带有 HotSpot Client JVM 的 Sun SDK。

② 对于生产模式，使用 BEA JRockit® SDK。该 SDK 提供最优运行性能，但与其他 SDK 相比，其初次启动周期可能需要更长的时间。

（2）更改 JAVA_VENDOR 变量的值。

例如，(setDomainEnv.cmd) for WINDOWS OS；

set JAVA_HOME=C:\myJDKs\jdk1.6.0_18；

set JAVA_VENDOR=Sun。

（3）重新启动当前运行的所有服务器。

8.9 WebLogic 如何打补丁

一般安装完产品后，可能要做的一件事情就是通过打补丁来修正 WebLogic 软件的一些 Bug，保障系统的稳定性。这一小节我们将简述 WebLogic Server 打补丁的步骤。

（1）如果您有 Oracle 产品的支持服务，建议您首先联系 Oracle 客户支持部门，获得相关技术支持。

（2）访问 Oracle 官网检查存在的补丁包。

（3）下载需要的最新的补丁包。

（4）减压补丁包后，找到其包含的 readme 文件，里面会有详细的操作步骤指导，一般情况下只要按照这个文档建议的步骤做就行了。

（5）按照 readme 文档的操作步骤打补丁。

（6）重启启动服务器。

第 9 章　与开源 SSH 框架的兼容

9.1　MVC 模型

9.1.1　MVC 简介

MVC（Modal View Controller，模型－视图－控制器）本来是存在于 Desktop 程序中的，M 是指数据模型，V 是指用户界面，C 则是控制器。使用 MVC 的目的是将 M 和 V 的实现代码分离，从而使同一个程序可以使用不同的表现形式。比如一批统计数据可以分别用柱状图、饼图来表示。C 存在的目的则是确保 M 和 V 的同步，一旦 M 改变，V 应该同步更新。

MVC 是 Xerox PARC 在 20 世纪 80 年代为编程语言 Smalltalk-80 发明的一种软件设计模式，至今已被广泛使用。最近几年被推荐为 Sun 公司 J2EE 平台的设计模式，并且受到越来越多使用 ColdFusion 和 PHP 的开发者的欢迎。MVC 模式是一个有用的工具箱，它有很多好处，但也有一些缺点。

9.1.2　MVC 如何工作

MVC 是一个设计模式，它强制性地使应用程序的输入、处理和输出分开。使用 MVC 应用程序被分成 3 个核心部件：模型、视图、控制器。它们各自处理自己的任务。

1．视图 V

视图是用户看到并与之交互的界面。对老式的 Web 应用程序来说，视图就是由 HTML 元素组成的界面，在新式的 Web 应用程序中，HTML 依旧在视图中扮演着重要的角色，但一些新的技术已层出不穷，它们包括 Macromedia Flash 和像 XHTML、XML/XSL、WML 等一些标识语言和 Web Services。如何处理应用程序的界面变得越来越有挑战性。MVC 一个大的好处是它能为您的应用程序处理很多不同的视图。在视图中其实没有真正的处理发生，不管这些数据是联机存储的，还是一个雇员列表，作为视图来讲，它只是作为一种输出数据并允许用户操纵的方式。

2．模型 M

模型表示企业数据和业务规则。在 MVC 的 3 个部件中，模型拥有最多的处理任务。例如，它可能用像 EJBs 和 ColdFusion Components 这样的构件对象来处理数据库。被模型返回的数据是中立的，就是说模型与数据格式无关，这样一个模型能为多个视图提供数据。

由于应用于模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性。

3．控制器 C

控制器接受用户的输入并调用模型和视图去完成用户的需求。所以当单击 Web 页面中的超链接和发送 HTML 表单时，控制器本身不输出任何东西和做任何处理。它只是接收请求并决定调用哪个模型构件去处理请求，然后再确定用哪个视图来显示模型处理返回的数据。

现在我们总结 MVC 的处理过程，首先控制器接收用户的请求，并决定应该调用哪个模型来进行处理，然后模型用业务逻辑来处理用户的请求并返回数据，最后控制器用相应的图格式化模型返回的数据，并通过表示层呈现给用户。

9.1.3　为什么要使用 MVC（优点）

大部分 Web 应用程序都是用像 ASP、PHP、JSP 或者 CFML 这样的过程化语言来创建的。它们将像数据库查询语句这样的数据层代码和像 HTML 这样的表示层代码混在一起。经验比较丰富的开发者会将数据从表示层分离开来，但这通常不是很容易做到的，它需要精心的计划和不断的尝试。MVC 从根本上强制性地将它们分开。

尽管构造 MVC 应用程序需要一些额外的工作，但是它给我们带来的好处是毋庸置疑的。

首先，最重要的一点是多个视图能共享一个模型，正如前面所提及的，现在需要用越来越多的方式来访问您的应用程序。对此，其中一个解决之道就是使用 MVC，无论您的用户想要 Flash 界面或是 WAP 界面，用一个模型就能处理它们。由于您已经将数据和业务规则从表示层分开，所以您可以最大化地重用您的代码。由于模型返回的数据没有进行格式化，所以同样的构件能被不同界面使用。

例如，很多数据可能用 HTML 来表示，但是它们也有可能要用 Macromedia Flash 和 WAP 来表示。模型也有状态管理和数据持久性处理的功能，例如，基于会话的购物车和电子商务过程也能被 Flash 网站或者无线联网的应用程序所重用。因为模型是自包含的，并且与控制器和视图相分离，所以很容易改变您的应用程序的数据层和业务规则。如果您想把您的数据库从 MySQL 移植到 Oracle，或者改变您的基于 RDBMS 数据源到 LDAP，只需改变您的模型即可。一旦您正确地实现了模型，不管您的数据来自数据库或是 LDAP 服务器，视图将会正确地显示它们。由于运用 MVC 的应用程序的 3 个部件是相互对立的，改变其中一个不会影响其他两个，所以依据这种设计思想您能构造良好的松耦合的构件。

另外，控制器也提供了一个好处，就是可以使用控制器来联接不同的模型和视图去完成用户的需求，这样控制器可以为构造应用程序提供强有力的手段。给定一些可重用的模型和视图，控制器可以根据用户的需求选择模型进行处理，然后选择视图将处理结果显示给用户。

9.1.4　MVC 的缺点

MVC 的缺点是由于它没有明确的定义，所以完全理解 MVC 并不是很容易。使用 MVC

需要精心的计划，由于它的内部原理比较复杂，所以需要花费一些时间去思考。您将不得不花费相当可观的时间去考虑如何将 MVC 运用到您的应用程序，同时由于模型和视图要严格地分离，这样也给调试应用程序带来了一定的困难。每个构件在使用之前都需要经过彻底的测试。一旦您的构件经过了测试，您就可以毫无顾忌的重用它们了。

根据经验，由于我们将一个应用程序分成了 3 个部件，所以使用 MVC 同时也意味着您将要管理比以前更多的文件，这一点是显而易见的。这样好像我们的工作量增加了，但是请记住这比起它所能带给我们的好处是不值一提的。

MVC 并不适合小型甚至中等规模的应用程序，花费大量时间将 MVC 应用到规模并不是很大的应用程序通常会得不偿失。MVC 设计模式是一个很好创建软件的途径，它所提倡的一些原则，像内容和显示互相分离可能比较好理解。但是如果您要隔离模型、视图和控制器的构件，您可能需要重新思考您的应用程序，尤其是应用程序的构架方面。如果您肯接受 MVC，并且有能力应付它所带来的额外的工作和复杂性，MVC 将会使您的软件在健壮性、代码重用和结构方面上一个新的台阶。

9.2 开源框架综述

9.2.1 Struts 简介

Struts 是 Apache 基金会 Jakarta 项目组的一个 Open Source 项目，它采用 MVC 模式，能够很好地帮助 Java 开发者利用 J2EE 开发 Web 应用。和其他的 Java 架构一样，Struts 也是面向对象设计的，将 MVC 模式“分离显示逻辑和业务逻辑”的能力发挥得淋漓尽致。Struts 框架的核心是一个弹性的控制层，基于如 Java Servlets、JavaBeans、ResourceBundles 与 XML 等标准技术，以及 Jakarta Commons 的一些类库。Struts 由一组相互协作的类（组件）、Servlet 以及 jsp tag lib 组成。基于 Struts 构架的 Web 应用程序基本上符合 JSP Model2 的设计标准，可以说是一个传统 MVC 设计模式的一种变化类型。

1. Model 部分

由 JavaBean 组成，ActionForm 用于封装用户的请求参数，封装成 ActionForm 对象，该对象被 ActionServlet 转发给 Action，Action 根据 ActionForm 里面的请求参数处理用户的请求。

JavaBean 则封装了底层的业务逻辑，包括数据库访问等。

2. View 部分

该部分采用 JSP 实现。

Struts 提供了丰富的标签库，通过标签库可以减少脚本的使用，自定义的标签库可以实现与 Model 的有效交互，并增加了现实功能。

3. Controller 组件

Controller 组件有两个部分组成，即系统核心控制器和业务逻辑控制器。

系统核心控制器，对应 Struts 中的 ActionServlet。该控制器由 Struts 框架提供，继承 HttpServlet 类，因此可以配置成标注的 Servlet。该控制器负责拦截所有的 HTTP 请求，然后根据用户请求决定是否要转给业务逻辑控制器。

业务逻辑控制器，负责处理用户请求，本身不具备处理能力，而是调用 Model 来完成处理。对应 Action 部分。

9.2.2　Spring 简介

Spring 是一个开源框架，它由 Rod Johnson 创建。它是为了解决企业应用开发的复杂性而创建的。Spring 使用基本的 JavaBean 来完成以前只可能由 EJB 完成的事情。然而，Spring 的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何 Java 应用都可以从 Spring 中受益。

目的：解决企业应用开发的复杂性。

功能：使用基本的 JavaBean 代替 EJB，并提供了更多的企业应用功能。

范围：任何 Java 应用。

简单来说，Spring 是一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架。

轻量——从大小与开销两方面而言 Spring 都是轻量的。完整的 Spring 框架可以在一个大小只有 1MB 多的 JAR 文件里发布。并且 Spring 所需的处理开销也是微不足道的。此外，Spring 是非侵入式的。典型地，Spring 应用中的对象不依赖于 Spring 的特定类。

控制反转——Spring 通过一种称做控制反转（IoC）的技术促进了松耦合。当应用了 IoC，一个对象依赖的其他对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。您可以认为 IoC 与 JNDI 相反，不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

面向切面——Spring 提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的，完成业务逻辑，仅此而已。它们并不负责（甚至是意识）其他的系统级关注点，例如，日志或事务支持。

容器——Spring 包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，您可以配置您的每个 bean 如何被创建。基于一个可配置原型（prototype），您的 bean 可以创建一个单独的实例或者每次需要时都生成一个新的实例以及它们是如何相互关联的。然而，Spring 不应该被混同于传统的重量级的 EJB 容器，它们经常是庞大与笨重的，难以使用。

框架——Spring 可以将简单的组件配置、组合成为复杂的应用。在 Spring 中，应用对象被声明式地组合，典型的应用是在一个 XML 文件里。Spring 也提供了很多基础功能（事务管理、持久化框架集成等），将应用逻辑的开发留给了用户。

所有 Spring 的这些特征使您能够编写更干净、更可管理、并且更易于测试的代码。它们也为 Spring 中的各种模块提供了基础支持。

1．起源

您可能正在想 Spring 不过是另外一个的 framework。当已经有许多开放源代码（和专

有）J2EE Framework 时，我们为什么还需要 Spring Framework？

Spring 是独特的，因为若干个原因。

它定位的领域是许多其他流行的 Framework 没有的。Spring 关注提供一种方法管理你的业务对象。

Spring 是全面的和模块化的。Spring 有分层的体系结构，这意味着您能选择使用它孤立的任何部分，它的架构仍然是内在稳定的。因此从您的学习中，您可以得到最大的价值。例如，你可能选择仅仅使用 Spring 来简单化 JDBC 的使用，或用来管理所有的业务对象。

它的设计从底部帮助您编写易于测试的代码。Spring 是用于测试驱动工程的理想的 Framework。

Spring 对您的工程来说，它不需要一个以上的 Framework。Spring 是潜在的一站式解决方案，定位于与典型应用相关的大部分基础结构中。它也涉及到其他 Framework 没有考虑到的内容。

2. 特点

（1）方便解耦，简化开发

通过 Spring 提供的 IoC 容器，我们可以将对象之间的依赖关系交由 Spring 进行控制，避免硬编码所造成的过度程序耦合。有了 Spring，用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

（2）AOP 编程的支持

通过 Spring 提供的 AOP 功能，方便进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应付。

（3）声明式事务的支持

在 Spring 中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

（4）方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，在 Spring 里，测试不再是昂贵的操作，而是随手可做的事情。

（5）方便集成各种优秀框架

Spring 不排斥各种优秀的开源框架，相反，Spring 可以降低各种框架的使用难度，Spring 提供了对各种优秀框架（如 Struts、Hibernate、Hession、Quartz 等）的直接支持。

（6）降低 Java EE API 的使用难度

Spring 对很多难用的 Java EE API（如 JDBC、JavaMail、远程调用等）提供了一个薄薄的封装层，通过 Spring 的简易封装，这些 Java EE API 的使用难度大大降低。

9.2.3 Hibernate 简介

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲的使用对象编程思维来操纵数据库。Hibernate 可以应用在任何使用 JDBC 的场合，既可以在 Java 的客户端程序使用，也可以在 Servlet/JSP 的

Web 应用中使用，最具革命意义的是，Hibernate 可以在应用 EJB 的 J2EE 架构中取代 CMP，完成数据持久化的重任。

Hibernate 的核心接口一共有 5 个，分别为 Session、SessionFactory、Transaction、Query 和 Configuration。这 5 个核心接口在任何开发中都会用到。通过这些接口，不仅可以对持久化对象进行存取，还能够进行事务控制。下面对这 5 个核心接口分别加以介绍。

1. Session 接口

Session 接口负责执行被持久化对象的 CRUD 操作（CRUD 的任务是完成与数据库的交流，包含了很多常见的 SQL 语句）。但需要注意的是 Session 对象是非线程安全的。同时，Hibernate 的 Session 不同于 JSP 应用中的 HttpSession。这里当使用 Session 这个术语时，其实指的是 Hibernate 中的 Session，而以后会将 HttpSession 对象称为用户 Session。

2. SessionFactory 接口

SessionFactory 接口负责初始化 Hibernate。它充当数据存储源的代理，并负责创建 Session 对象。这里用到了工厂模式。需要注意的是 SessionFactory 并不是轻量级的，因为一般情况下，一个项目通常只需要一个 SessionFactory 就够了，当需要操作多个数据库时，可以为每个数据库指定一个 SessionFactory。

3. Configuration 接口

Configuration 接口负责配置并启动 Hibernate，创建 SessionFactory 对象。在 Hibernate 的启动的过程中，Configuration 类的实例首先定位映射文档位置、读取配置，然后创建 SessionFactory 对象。

4. Transaction 接口

Transaction 接口负责事务相关的操作。它是可选的，开发人员也可以设计编写自己的底层事务处理代码。

5. Query 和 Criteria 接口

Query 和 Criteria 接口负责执行各种数据库查询。它可以使用 HQL 语言或 SQL 语句两种表达方式。

9.3　WebLogic 与 Spring 的兼容性

企业 Spring：Spring Framework 的非侵入性 IoC 研发模型不仅依赖于对 J2EE 应用服务器可用的特性集，而且旨在补充该特性集。事实上，在苛刻的生产环境中，底层应用服务器基础架构所提供的服务质量对于 Spring 应用程式的可靠性、可用性和性能非常重要。WebLogic Server 9.0 所提供的企业级特性能增强了 Spring 应用程式的所有方面。

9.3.1 集群化部署 Spring 应用

一个 WebLogic Server 集群包括多个 WebLogic Server 服务器实例，这些服务器实例同时运行并一起工作，从而提高了可伸缩性和可靠性。对客户端来说是透明的，集群对外就像单个的 WebLogic Server 实例一样。构成集群的服务器实例既能运行在同一台机器上，也能位于不同的机器上。能通过在现有的机器上向集群添加另外的服务器实例，或向集群添加机器以驻留增加的服务器实例来提高集群的容量。

WebLogic Server 集群为 Spring 应用程式提供了一个企业级的部署平台，虽然其他的技术产品也支持类似的特性，不过它们不具有 WebLogic Server 所提供的丰富性和易用性。Spring 应用程式通常都被打包为 Web 应用程式，这种情况下，要利用 WebLogic Server 集群就无需修改应用程式。只要把应用程式部署到集群中的服务器上，就能获得增强的可伸缩性和可用性

9.3.2 Spring 会话复制

Spring Web 应用程式习惯在 HTTP 会话中保存信息，比如订单 ID 和用户信息。为了支持集群中 Servlet 和 JSP 的自动复制和故障恢复，WebLogic Server 支持几种用于保持 HTTP 会话状态的机制。只要为应用程式提供正确的 weblogic.xml 部署描述符，Spring Web 应用程式就能非侵入性地使用这些机制。

9.3.3 集群化的 Spring 远程控制

Spring 提供功能强大的远程控制支持，允许用户轻松导出和使用远程服务，同时仍然能利用基于 POJO 的一致编程模型。通过一个结合到适当的 Spring bean 的 RMI 接口，Vanilla Spring 支持代理 POJO 调用。然而，这种支持仅限于 JRMP（Sun 的 RMI 实现），或通过 JndiRmiProxyFactoryBean 使用特定的远程接口。

借助于 Spring on WebLogic Server 认证，扩展了 JndiRmiProxyFactoryBean 和相关的服务导出程式，这样它就能支持所有 J2EE RMI 实现的 POJO 代理，包括 RMI-IIOP 和 T3。这方面的支持还包括一个 WebLogic RMI 部署描述符，它支持代理 RMI 接口上的集群化，所以 POJO 调用能跨一个 WebLogic Server 集群进行负载均衡。集群化的描述符是自动包含在内的，只需要以适当方式设置集群和将 Spring 应用程式部署到所有集群成员中。

9.3.4 对 Spring 组件的控制台支持

Spring on WebLogic Server 工具包中包含一个 WebLogic Server 控制台扩展，他显示了定义在应用程式中的 Spring bean、属性和操作。它构建在 WebLogic 控制台扩展门户框架之上，该框架能变换 WebLogic Administration 控制台的外观、功能和布局，而无须修改服务器或控制台代码。将控制台扩展复制到 yourdomain/console-ext 目录下，则重新启动服务

器时就部署了控制台扩展（参考 Spring on WebLogic Server 工具包）。

该扩展自动为不是 MBean 的 Spring bean（大多数 Spring bean）创建（JMX）管理接口，然后在 applicationContext.xml 中设置一个 MbeanExporter，并指定哪些 bean 要通过该 exporter 公开，这样控制台扩展就运行了。这项特性是 Spring 和 WebLogic Server 进行无缝和非侵入性合作的一个良好例证。要使应用程式支持 JMX，只需修改应用程式上下文部署描述符。要使控制台支持 Spring，只需将一个简单的 JAR 部署到现有的域即可。

另外，WebLogic10.3 添加支持 Spring 的控制台扩展

（1）登录到管理控制台。

（2）在控制台中，单击工具栏上的“首选项”按钮，如图 9-1 所示。

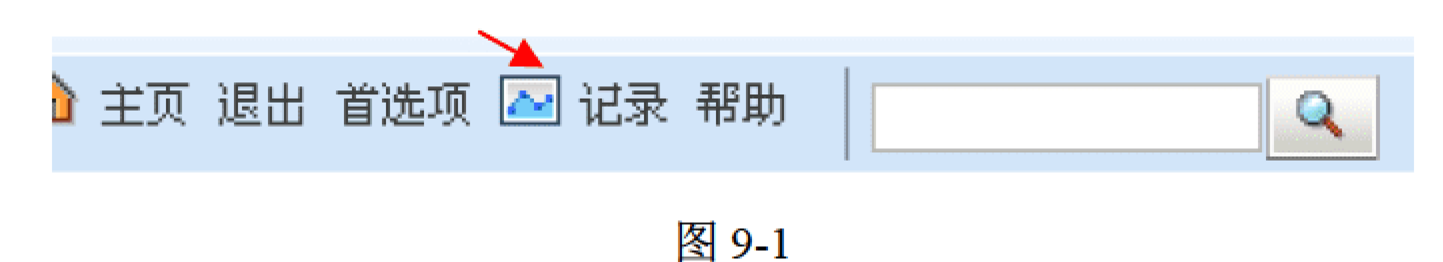


图 9-1

（3）在选项页单击“扩展”按钮，如图 9-2 所示。

（4）选中 spring-console 复选框，然后单击“启用”按钮，如图 9-3 所示。

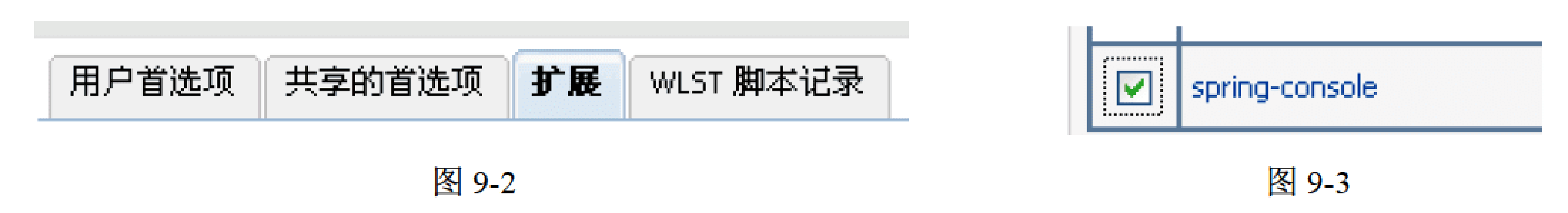


图 9-2　　图 9-3

（5）停止服务器，然后重新启动使更改生效。

9.3.5　Web 服务支持

Spring 远程控制功能的另一个方面是它对 RPC 风格 Web 服务的支持。WebLogic Server 提供了基于 Ant 的工具，用于基于 Web 服务的 WSDL 描述生成 JAX-RPC 存根。Web 服务客户端使用这些生成的存根来获取代表服务器端操作的一个远程接口。Spring 提供了一个 JaxRpcPortProxyFactoryBean 来简化这个过程。我们发现，在 WebLogic Server 环境中设置 JaxRpcPortProxyFactoryBean 有些棘手，所以为了节约客户的时间，我们给出下面这个代码片断，演示怎么为一个包含复杂类型的 Document Literal 风格的 Web 服务设置代理生成。

这里大部分属性都是自解释的，其中有一些属性比较重要。

（1）serviceInterface 是 Spring 的 setter 注入的副产品。这个类将表示 Web 服务操作。

（2）customProperties 属性支持制定的 WebLogic Server Web 服务存根属性。

（3）jaxRpcService 值被设置为 WebLogic Server 生成的 JAX-RPC 实现服务。JAX-RPC 服务负责验证 Web 服务和加载复杂的类型映射。为了实现后者，必须把 WebLogic Server 的 JAX-RPC 服务实现设置为 Spring bean。这确保了 JAX-RPC 服务构造函数的执行，这也是加载类型映射文件的地方。

把 JaxRpcPortProxyFactoryBean 上的 lookupServiceOnStartup 设置为 false，能关闭启动期间的 JAX-RPC 服务查找。这样，查找将在首次访问时进行。这对于和 WebLogic Server 的可靠请求/响应 Web 服务通信的客户端来说是必需的，而且此处的客户端也必须是个 Web 服务。通常在这些情况下，始发客户端是和 Web 服务客户端一起部署的。因为直到应用程式部署

完成才会激活 Web 服务，所以客户端 Web 服务对于 Spring 的上下文加载是不可用的。

9.3.6 安全性框架

WebLogic Server 安全系统支持和扩展了 J2EE 安全性，同时提供一组丰富的安全提供程式，您能对它们进行制定，然后使用它们来处理不同的安全性数据库或安全性策略。除了使用标准的 J2EE 安全性之外，应用程式程式员还能使用非常多专有扩展，这些扩展使应用程式能和安全系统紧密集成。WebLogic Server 带有几个安全提供程式，例如，能选择包含大部分流行 LDAP 服务器的身份验证数据库、Active Directory、本地视窗系统和一个内置的身份验证解决方案。能使用制定的提供程式对内置的提供程式进行扩充，从而几乎能和任意身份验证数据库、授权机制和凭证映射服务相集成。

因为部署为 webapp 的 Spring 应用程式使用的是 J2EE 安全性，所以无须修改应用程式就能获得 WebLogic Server 的安全性好处。

经验丰富的 Spring 用户还会熟悉 Acegi-Spring 自身的安全框架。目前，能在应用程式中使用 Acegi、WebLogic Server 安全性，或同时使用二者，因为它们是相互独立的。

9.3.7 分布式事务支持

Spring 为事务管理提供了基础架构。除了对各家数据库供给商提供支持之外，Spring 还通过一家 J2EE 供给商的 JTA 实现支持分布式事务。通过 WebLogic JTA Transaction Manager，能把 Spring 的 JTA 管理器设置为和 WebLogic Server 的 JTA，实现一起工作。

WebLogic JTA Transaction Manager 把责任直接委派给 WebLogic Server 的 Java Transaction API。WebLogic Server 的 JTA Transaction Manager 接口能通过 JNDI 为客户端和 bean 提供者所用，而由 Spring 来管理这种交互。事务管理器还支持事务的作用域，事务能作用于集群和域内部或二者之间。

WebLogic JTA Transaction Manager 最强大的特性是管理分布式事务的能力和用于企业应用程式的两阶段提交协议。通过采用 WebLogic JTA Transaction Manager，应用程式能通过 WebLogic Administration Console 来进行事务监视。WebLogic JTA Transaction Manager 还支持按数据库（per-database）隔离级别，这种级别支持复杂的事务设置。

9.3.8 WebLogic Server 上的 Spring Framework 版本兼容

1. WLS9.x（表 9-1）

表 9-1

更改请求编号	描述和变通方法或解决方案	找到位置	解决位置
CR242675	在 RMI 类加载器中发生了 NullPointerException 变通方法或解决方案： 可与 BEA 客户支持联系以获取 WebLogic Server/Spring 合并修补程序	9.0	9.2

续表

更改请求编号	描述和变通方法或解决方案	找到位置	解决位置
CR236708	在 Hibernate 3 和 WebLogic Server 之间存在 Antlr 冲突 变通方法或解决方案： 将 Antlr2.7.5.jar 放在 CLASSPATH 中的 weblogic.jar 之前	8.1SP05、9.0	9.2
CR242923	T3 运行时无法对包含基元类型的类描述符进行解码 变通方法或解决方案： 可与 BEA 客户支持联系以获取 WebLogic Server-Spring 合并修补程序	9.0	9.2
CR242883	IIOP 运行时无法对包含基元类型的类描述符进行解码 变通方法或解决方案： 可与 BEA 客户支持联系以获取 WebLogic Server-Spring 合并修补程序	9.0	9.2
CR237532	Spring Framework 存在 Web 应用程序类加载问题 变通方法或解决方案： 可与 BEA 客户支持联系以获取 WebLogic Server-Spring 合并修补程序	8.1SP05、9.0	9.2
CR241195	在 Spring Pet Clinic 示例应用程序中更新记录会导致以下错误：java.lang.IllegalStateException: Cannot access session scope since the requested page does not participate in a session. at weblogic.servlet.jsp.PageContextImpl.getAttribute(PageContextImpl.java:273) at javax.servlet.jsp.jstl.core.Config.get(Config.java:145) at javax.servlet.jsp.jstl.core.Config.find(Config.java:393) at org.apache.taglibs.standard.tag.common.fmt.TimeZoneSupport.getTimeZone(TimeZoneSupport.java:140) 变通方法或解决方案： 将 includes.jsp 文件中的第一行标记为注释	9.0	9.2
CR244683	HP-UX 需要 jdk150_01，而不是 jdk150_03 变通方法或解决方案： 在 medrec-spring 目录中，使用 jdk150_01 替换 jdk150_03	9.0	9.2
CR244693	当您从远程计算机上访问 MedRec-Spring 时，MedRec-Spring 退出功能不起作用 变通方法或解决方案： 不从远程计算机访问 MedRec-Spring 应用程序，并且不将 localhost 用于请求重定向	9.0	9.2
CR244691	对 WebLogic 管理控制台的 Spring 扩展仅支持 Web 应用程序(.war)文件，无法用于监视非.war 文件（如 MedRec-Spring）中的 Spring Bean	9.0	9.2
CR243957	使用 CTRL-C 关闭 WebLogic Server 时，如果正在破坏 bean domainMBeanServerConnection，则可能会发生关闭异常 变通方法或解决方案： 使用标志-Dweblogic.slc=true 以便确定启动和停止 domainRuntimeServerService 的时间	9.0	9.2

续表

更改请求编号	描述和变通方法或解决方案	找到位置	解决位置
CR280985	无法通过将 countries_mbeans.war 应用程序复制到 WebLogic Server 域目录的 autodeploy 目录来自动部署该应用程序 countries_mbeans.war Web 应用程序是一个 Spring 测试扩展应用程序 变通方法或解决方案： 使用 WebLogic Server 管理控制台来部署 countries_mbeans.war Web 应用程序，而不是自动部署	9.2	
CR301115	在 Spring Pet Clinic 示例应用程序中运行单元测试会导致以下错误： 从 weblogic.xml.jaxp.RegistrySAXTransformerFactory 中找不到有效的处理器版本实现 变通方法或解决方案： 通过将以下条目添加到 $java.home/lib/jaxp.properties 文件来定义 XML 解析器类 ❑ javax.xml.transform.TransformerFactory=org.apache.xalan.processor.TransformerFactoryImpl ❑ javax.xml.xpath.XPathFactory=org.apache.xpath.jaxp.XPathFactoryImpl ❑ javax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.SAXParserFactoryImpl ❑ javax.xml.parsers.DocumentBuilderFactory=org.apache.xerces.jaxp.DocumentBuilderFactoryImpl	9.2	
CR300748	访问部署到 WebLogic Server 9.2 的 tiles-samples 时会出现异常	9.2	

2．WLS10.x

在该版本上 Spring Framework 2.0.2 经过了官方认证，表 9-2。

表 9-2

更改请求编号	描述和变通方法或解决方案	找到位置	解决位置
CR319968	使用 JRockit 时，OpenJPA 的 ClassFileTranformer 工作不正常 方案： 在编译阶段进行 enhence 而不要使用 LoadTimeWeaver，在系统加载阶段 enhence	10.0	
CR320649	使用 JRockit 时，在 WebLogic 上部署 Spring Pet Clinic 样例程序失败，产生和 OpenJPA 相关的异常 方案： 将 JRockit 版本从 R26.4 升级到 R27.2	10.0	10.0

9.3.9　Spring 中遇到的问题

1．版本兼容性的问题

测试环境：JDK1.6、Spring 2.0.5、Spring 2.5，中间件为 WebLogic 10.3.1、WebLogic 8.1。

经过测试：Spring 2.0.5 在 WebLogic 8.1 表现良好。在部署的过程中没有出现什么问题。Spring 2.5 在 WebLogic 8.1 下就不能部署而且每次在报错，Spring 2.5 在 WebLogic 10.3.1 表现良好。在部署的过程中没有出现什么问题，Spring 2.0.5 在 WebLogic10.3.1 下也会出现报错现象而不能部署。

在 Oracle 的官方网站上可以找到下列内容

wls 10.3　支持 Spring 2.5.x。

wls 10.0　支持 Spring 2.0.x。

wls 9.2　 支持 Spring 1.2.8、2.0.x、2.5.x。

另外，当 Spring 的版本升级到 2.5.6 的时候还是出现了报错现象，错误提示如下。

示例 9-1：

```
12:23:59,656 ERROR XFireServlet:51 - Error initializing XFireServlet.
org.springframework.beans.factory.BeanDefinitionStoreException: Unexpected
exception parsing XML document from class path resource [org/codehaus/xfire/
spring/xfire.xml];
nested exception is java.lang.ClassCastException:
weblogic.xml.jaxp.RegistryDocumentBuilderFactory cannot be cast to javax.
xml.parsers.DocumentBuilderFactory
```

这个错误提示是说 WebLogic 的 RegistryDocumentBuilderFactory 类和 Java 中 xml.parsers.DocumentBuilderFactory 类在编译的时候发生冲突。

这样的错误信息提示找到项目中的冲突包移除就没有什么了，在项目中移除两个 jar 包：wstx-asl-3.2.0.jar 和 xml-apis-1.0.b2.jar，这样项目就能够成功部署了。

2．项目中出现的 nullbean 问题

这个问题主要是 Spring 的框架中的监听器没有加载，也就是在 web.xml 加载的时候就失败了。Spring 为 web.xml 中加载监听提供了两种方式，但是达到的效果是一样的。

首先介绍 Spring 的监听器作用。ContextLoaderListener 的作用就是启动 Web 容器时，自动装配 ApplicationContext 的配置信息。因为它实现了 ServletContextListener 这个接口，在 web.xml 配置这个监听器，启动容器时，就会默认执行它实现的方法。在 ContextLoaderListener 中关联了 ContextLoader 这个类，所以整个加载配置过程由 ContextLoader 来完成。ContextLoader 创建的是 XmlWebApplicationContext 这样一个类，它实现的接口是 WebApplicationContext→ConfigurableWebApplicationContext→ApplicationContext→BeanFactory，这样一来 Spring 中的所有 bean 都由这个类来创建。

如果在 web.xml 中不写任何参数配置信息，默认的路径是/WEB-INF/applicationContext.xml，在 WEB-INF 目录下创建的 xml 文件的名称必须是 applicationContext.xml。如

果是要自定义文件名，可以在 web.xml 里加入 contextConfigLocation 这个 context 参数。

示例 9-2

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/classes/applicationContext-*.xml
  </param-value>
</context-param>
```

在<param-value> </param-value>里指定相应的 xml 文件名，如果有多个 xml 文件，可以写在一起并以“，”分隔。上面的 applicationContext-*.xml 采用通配符，比如这目录下有 applicationContext-ibatis-base.xml,applicationContext-action.xml,applicationContext-ibatisdao.xml 等文件，都会一同被载入。

9.4 WebLogic 与 Struts 的兼容性

9.4.1 调试和日志记录 Struts 应用程序

WebLogic 提供了它自己的日志子系统，该系统是 WebLogic 的缺省日志框架。不过，也可以将 WebLogic 与其他日志框架（如 Log4j）一起使用。默认情况下，Struts 将把所有消息记录到 WebLogic 日志框架中。要在开发 Struts 应用程序过程中进行最有效的记录，需要确保将 Logging Message 严重性级别设置为 INFO，这样将记录所有 INFO 级别以上的日志，从而可以在出错时进行有效的调试。

还建议将所有 system.out.println()消息都重定向到 WebLogic 日志中，以方便调试。可以指定记录这些消息的 stdout 文件。只需编辑 WebLogic Server 脚本，使 JAVA_OPTIONS 变量做以下指定：

-Dweblogic.Stdout="stdout-filename"

-Dweblogic.Stderr="stderr-filename"

要查看所有发出的 HTTP 请求的日志，可以参考 WebLogic Server 生成的访问日志（access log）。该日志提供了访问请求的一些详细信息，比如时间戳、访问的 URL 和 HTTP 返回代码等。

示例 9-3：

```
127.0.0.1 - - [13/Jul/2004:21:39:46 -0615] "GET /struts-sample/
preregisterCab.do?IndiReport=true HTTP/1.1" 200 4173
127.0.0.1 - - [13/Jul/2004:21:41:14 -0615] "POST /struts-sample/
getIndiData.do;jsessionid=A05R5lY5jLRJokg8BRMYgLwCOvfuVzC
7KEE4AsCZijqD7l6A22Re!-237055073 HTTP/1.1" 200 4667
127.0.0.1 - - [16/Jul/2004:00:07:58 -0615] "GET /struts-sample/
```

```
preregisterCab.do?IndiReport
=true HTTP/1.1" 200 4173
127.0.0.1 - - [16/Jul/2004:00:08:26 -0615] "POST /struts-sample/
getIndiData.do;jsessionid=A32LikMTeXrS5Bnbp7p1TASmcLHkuko9Sudr8Lja
tWw2AqeP6zkt!1759102893 HTTP/1.1" 200 8893
127.0.0.1 - - [19/Jul/2004:04:39:36 -0615] "GET /struts-sample/
preregisterCab.do?IndiReport
=true HTTP/1.1" 200 4173
127.0.0.1 - - [19/Jul/2004:04:40:16 -0615] "POST /struts-sample/
getIndiData.do;jsessionid=A7yxT3KtTy01T7A39DLSup2RPqxvV9uWay325WxB
Lqtj1dlx3Ewa!-431463598 HTTP/1.1" 200 5416
127.0.0.1 - - [14/Aug/2004:22:36:24 -0615] "GET /struts-sample/
preregisterCab.do?IndiRepor
t=true HTTP/1.1" 200 4173
127.0.0.1 - - [14/Aug/2004:22:36:58 -0615] "POST /struts-sample/
getIndiData.do;jsessionid=Be2WKm1nP7DGdX1uIY8PraQdx51wBZTjVSswGHdcbL4njcj
XOdJh!1670519755 HTTP/1.1" 200 4684
127.0.0.1 - - [14/Aug/2004:22:36:59 -0615] "GET /struts-sample/
pages/STFull.jpg HTTP/1.1" 304 0
```

9.4.2　调试 WebLogic 类加载器

在开发 Struts 应用程序的过程中，必然会遇到类加载器问题。调试那些 NoClassDefFoundError 和 ClassNotFoundException 类的确是一件麻烦事。无法在父类加载器和子类加载器中找到类时，会发生 ClassNotFoundException，这很可能是打包问题造成的。加载了请求的类但无法找到依赖类时，会抛出 NoClassDefFoundError。父类加载器中的类从不引用子类加载器中的类。类加载器会先请求它们的父类加载器加载类，然后才会尝试自行加载类，因此在此类情况下可能会遇到 NoClassDefFoundError。

为调试 WebLogic 类加载器，WebLogic 提供了类加载器特定的调试标志。设置这些调试标志的方法如下。

（1）编辑 StartWeblogic 脚本，添加以下内容作为启动 WebLogic 时的命令行参数。Dweblogic.Debug = debug.lineNumbers,debug.methodNames,weblogic.ClassLoaderVerbose, weblogic.ClassLoader。

（2）确保在控制台的 Logging 选项中将 Debug to Stdout 选项设置为 enabled。

启用这些调试选项后，您就会注意到当应用程序加载任何类时都将有相应的类似，如下的日志所示。

示例 9-4：

```
[GenericClassLoader] : Looking for class: org.apache.struts.taglib.html.
ImgTag...
 With classpath of : ()
Classloader object id (weblogic.utils.classloaders.GenericClassLoader@
```

```
10d3f0d finder: weblogic.utils.classloaders.MultiClassFinder@1510d96
annotation: )
 [GenericClassLoader] : Class org.apache.struts.taglib.html.ImgTag not
 found.
 [GenericClassLoader] : Looking for class: org.apache.struts.taglib.html.
 ImgTag...
 With classpath of : ()
Classloader object id (weblogic.utils.classloaders.GenericClassLoader@
1ce64f6 finder: weblogic.utils.classloaders.MultiClassFinder@52fecf
annotation: ApplicationClassLoader@)
 [GenericClassLoader] : Class org.apache.struts.taglib.html.ImgTag not
 found.
 [GenericClassLoader] : Looking for class: org.apache.struts.taglib.
 html.ImgTag...
 With classpath of : (C:\APP-INF\classes)
 Classloader object id (weblogic.utils.classloaders.GenericClassLoader@
 1d6d61d finder: weblogic.utils.classloaders.MultiClassFinder@d6ee28
 annotation: struts-example@)
 [GenericClassLoader] : Class org.apache.struts.taglib.html.ImgTag not
 found.
 [ChangeAwareClassLoader] : weblogic.utils.classloaders.ChangeAwareClass-
 Loader@18a6890 finder: weblogic.utils.classloaders.MultiClassFinder@
 ad8bb4  annotation: struts-example@struts-example about to loadClass(org.
 apache.
struts.taglib.html.ImgTag)
 [GenericClassLoader] : Looking for class: org.apache.struts.taglib.
 html.ImgTag...
```

上例中 Struts 应用程序寻找的是 org.apache.struts.taglib.html.ImgTag 类。首先在父类加载器中寻找该类，然后在应用程序加载器中寻找，最后在 war 类加载器中寻找。如果在类加载器中找不到某个类，您会注意到日志中产生了下面这样的消息：

示例 9-5：

```
 [GenericClassLoader] : Looking for class: org.apache.struts.taglib.
html.ImgTag...
 With classpath of : ()
Classloader object id (weblogic.utils.classloaders.GenericClassLoader@
10d3f0d finder: weblogic.utils.classloaders.MultiClassFinder@1510d96
annotation: )
 [GenericClassLoader] : Class org.apache.struts.taglib.html.ImgTag not
 found.
 [GenericClassLoader] : Looking for class: org.apache.struts.taglib.
 html.ImgTag...
 With classpath of : ()
Classloader object id (weblogic.utils.classloaders.GenericClassLoader@
```

```
1ce64f6 finder: weblogic.utils.classloaders.MultiClassFinder@52fecf
annotation: ApplicationClassLoader@)
 [GenericClassLoader] : Class org.apache.struts.taglib.html.ImgTag not
 found.
 [GenericClassLoader] : Looking for class: org.apache.struts.taglib.
 html.ImgTag...
 With classpath of : (C:\APP-INF\classes)
 Classloader object id (weblogic.utils.classloaders.GenericClassLoader@
1d6d61d finder: weblogic.utils.classloaders.MultiClassFinder@d6ee28
annotation: struts-example@)
 [GenericClassLoader] : Class org.apache.struts.taglib.html.ImgTag not
 found.
 [ChangeAwareClassLoader] : weblogic.utils.classloaders.ChangeAware-
ClassLoader@18a6890 finder: weblogic.utils.classloaders.MultiClassFinder@
ad8bb4 annotation: struts-example@struts-example about to loadClass(org.
apache.struts.taglib.html.ImgTag)
 [GenericClassLoader] : Looking for class: org.apache.struts.taglib.html.
 ImgTag...
 With classpath of : (C:\bea-plat-81sp3\user_projects\domains\
sqlservdomain\myserver\.wlnotdelete\extract\myserver_struts-example_str
uts-example\jarfiles\WEB-INF\lib\log4j.jar;C:\bea-plat-81sp3\user proje
cts\domains\sqlservdomain\myserver\.wlnotdelete\extract\myserver_struts-
example_struts-example\jarfiles\WEBINF\lib\log4j-core.jar)
 Classloader object id (weblogic.utils.classloaders.ChangeAwareClass-
 Loader@18a689
0 finder: weblogic.utils.classloaders.MultiClassFinder@ad8bb4 annotation:
struts-example@struts-example)
 [GenericClassLoader] : Class org.apache.struts.taglib.html.ImgTag not
 found.
 <Nov 17, 2004 4:37:52 PM PST> <Error> <HTTP> <BEA-101017> <[ServletContext
(id=28746180,name=struts-example,context-path=/struts-example)] Root
cause of ServletException
java.lang.NoClassDefFoundError: org/apache/struts/taglib/html/ImgTag
at jsp_servlet.__index._jspService(__index.java:434)
 at weblogic.servlet.jsp.JspBase.service(JspBase.java:33)
at  weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction.run
(ServletStubImpl.java:996)
 at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(Servlet-StubImpl.java:419)
 at weblogic.servlet.internal.ServletStubImpl.invokeServlet
 (ServletStubImpl.java:463)
 at weblogic.servlet.internal.ServletStubImpl.invokeServlet
 (ServletStubImpl.java:315)
```

```
 at weblogic.servlet.internal.WebAppServletContext$ServletInvocation-
Action.run
 (WebAppServletContext.java:6452)
at  weblogic.security.acl.internal.AuthenticatedSubject.doAs(Authenticate
dSubject.java:321)
```

9.5 WebLogic 与 Hibernate 的兼容性

9.5.1 Hibernate 中可能遇到的问题

系统抛出下边的异常，导致 WebLogic 异常退出。

示例 9-6：

```
org.hibernate.hql.ast.HqlToken  org.hibernate.QueryException: ClassNot-
FoundException: org.hibernate.hql.ast.HqlToken [from com.dhcc.mm5.QbMm5-
List]
  at org.hibernate.hql.ast.HqlLexer.panic(HqlLexer.java:57)
  at antlr.CharScanner.setTokenObjectClass(CharScanner.java:340)
  at org.hibernate.hql.ast.HqlLexer.setTokenObjectClass(HqlLexer.java:31)
  at antlr.CharScanner.<init>(CharScanner.java:51)
  at antlr.CharScanner.<init>(CharScanner.java:60)
  at org.hibernate.hql.antlr.HqlBaseLexer.<init>(HqlBaseLexer.java:56)
  at org.hibernate.hql.antlr.HqlBaseLexer.<init>(HqlBaseLexer.java:53)
  at org.hibernate.hql.antlr.HqlBaseLexer.<init>(HqlBaseLexer.java:50)
  at org.hibernate.hql.ast.HqlLexer.<init>(HqlLexer.java:26)
  at org.hibernate.hql.ast.HqlParser.getInstance(HqlParser.java:44)
  at org.hibernate.hql.ast.QueryTranslatorImpl.parse(QueryTranslatorImpl.
  java:232)
  at org.hibernate.hql.ast.QueryTranslatorImpl.doCompile(QueryTrans-
  latorImpl.java:155)
  at org.hibernate.hql.ast.QueryTranslatorImpl.compile(QueryTranslator-
  Impl.java:109)
  at org.hibernate.engine.query.HQLQueryPlan.<init>(HQLQueryPlan.java:75)
  at org.hibernate.engine.query.HQLQueryPlan.<init>(HQLQueryPlan.java:54)
  at org.hibernate.engine.query.QueryPlanCache.getHQLQueryPlan(QueryPlan-
  Cache.java:71)
  at org.hibernate.impl.AbstractSessionImpl.getHQLQueryPlan(Abstract-
  SessionImpl.java:133)
  at org.hibernate.impl.AbstractSessionImpl.createQuery(Abstract-
  SessionImpl.java:112)
  at org.hibernate.impl.SessionImpl.createQuery(SessionImpl.java:1583)
```

```
    at com.dhcc.mm5.QbMm5ListDAO.findAll(QbMm5ListDAO.java:141)
    at jsp_servlet.__left._jspService(__left.java:114)
    at weblogic.servlet.jsp.JspBase.service(JspBase.java:34)
    at weblogic.servlet.internal.StubSecurityHelper$ServletServiceAction.
    run(StubSecurityHelper.java:226)
    at weblogic.servlet.internal.StubSecurityHelper.invokeServlet(Stub-
    SecurityHelper.java:124)
    at weblogic.servlet.internal.ServletStubImpl.execute(ServletStubImpl.
    java:283)
    at weblogic.servlet.internal.ServletStubImpl.onAddToMapException
    (ServletStubImpl.java:391)
    at weblogic.servlet.internal.ServletStubImpl.execute(ServletStubImpl.
    java:309)
    at weblogic.servlet.internal.ServletStubImpl.execute(ServletStubImpl.
    java:175)
    at weblogic.servlet.internal.WebAppServletContext$ServletInvocation-
    Action.run(WebAppServletContext.java:3370)
    at weblogic.security.acl.internal.AuthenticatedSubject.doAs(Authenti-
    catedSubject.java:321)
    at weblogic.security.service.SecurityManager.runAs(Unknown Source)
    at weblogic.servlet.internal.WebAppServletContext.securedExecute(WebApp-
    ServletContext.java:2117)
    at weblogic.servlet.internal.WebAppServletContext.execute(WebAppServlet-
    Context.java:2023)
    at weblogic.servlet.internal.ServletRequestImpl.run(ServletRequestImpl.
    java:1359)
    at weblogic.work.ExecuteThread.execute(ExecuteThread.java:200)
    at weblogic.work.ExecuteThread.run(ExecuteThread.java:172)
```

9.5.2　问题原因分析

Hibernate 3.0 采用新的基于 ANTLR 的 HQL/SQL 查询翻译器，然而在 weblogic.jar 中已经包含了 antrl 类库，但是版本并不一致，因此就会产生一些类加载的错误。出现这个错误之后，antlr 会调用 System.exit()，这样就会导致 WebLogic 中止服务。

在 Hibernate 的配置文件中，hibernate.query.factory_class 属性用来选择查询翻译器。

（1）选择 Hibernate 3.0 的查询翻译器：

hibernate.query.factory_class= org.hibernate.hql.ast.ASTQueryTranslatorFactory

（2）选择 Hibernate 2.1 的查询翻译器：

hibernate.query.factory_class= org.hibernate.hql.classic.ClassicQueryTranslatorFactory

为了使用 3.0 的批量更新和删除功能，只能选择（1），否则不能解释批量更新的语句，当使用的时候出现了不支持条件输入中文的情况，选择（2）可以支持输入中文，但没法解释批量更新语句。

9.5.3 解决方法

在 hibernate.properties 文件中增加属性：hibernate.query.factory_class，属性的值是 org.hibernate.hql.classic.ClassicQueryTranslatorFactory。如果用的是 cfg.xml 文件，就在 hibernate.cfg.xml 中的<session-factory>进行测试。

下面添加一条声明：

<property name="query.factory_class">org.hibernate.hql.classic.ClassicQueryTranslatorFactory </property>

问题解决了。另外，可以修改 startWebLogic.cmd，将

set CLASSPATH=%WEBLOGIC_CLASSPATH%;%POINTBASE_CLASSPATH%;%JAVA_HOME%\jre\lib\rt.jar;%WL_HOME%\server\lib\webservices.jar;%CLASSPATH%

改为

set CLASSPATH=%D:\wwwportalRun\WEB-INF\lib\antlr-2.7.5H3.jar;%WEBLOGIC_CL ASSPATH%;%POINTBASE_CLASSPATH%;%JAVA_HOME%\jre\lib\rt.jar;%WL_HOME%\server\lib\webservices.jar;%CLASSPATH%即可。

小结：以上主要提及了 WebLogic 与 Spring 和 Hibernate 的兼容性问题，一般情况下 WebLogic 与 SSH 兼容问题在于类（包）冲突，由于 WebLogic 自带了 classloader，会首先去找到自己的 jar 文件，然后加载项目的 jar，很多项目中的 jar 均已在 WebLogic 的 jar 中被加载了，这会直接导致无法在项目的 war 或者 ear 中找到 jar 文件。通常的解决办法就是在 classpath 中进行配置，让需要的包在 webservice.jar 之前来加载。

9.6 从 Tomcat 开源项目移植入 WebLogic 问题总结

9.6.1 JDK 和 Servlet 版本问题

WebLogic 8.1 sp4 以前（包括 sp4）只支持 JDK1.4，建议使用 JDK1.4 进行编译代码，有时 JDK1.5 编译的程序无法运行。由于 WebLogic 8.1 不支持 J2EE1.4，因此，不要使用 Servlet2.4 和 JSP2.0 进行编码。

9.6.2 Include 问题

在 BEA WebLogic 中不允许在一个文件中出现一次以上类似<%@ page contentType="text/html; charset=GBK"%>的代码，所以使用 include file 时，请将被 include 的文件中类似的代码删除。

在 Tomcat 时允许上述代码出现多次，并且使用 include file 时，被 include 的文件中不包含上述代码，编译后客户端显示为乱码。BEA 对此解释为 Tomcat 不符合 J2EE 规范。

为了增加代码的通用性和可移植性，建议使用<jsp:include>方式。

<jsp:include>将被 include 的 Jsp 代码视为独立存在的文件，所以可以在不同文件内使

用多个<%@ page contentType="text/html; charset=GBK"%>。<jsp:include>直接传参由<jsp:param>标签完成，在被 include 页面可以通过 request 得到传入的值，也可以通过 request.setAttribute()、request.getAttribute()进行内外文件参数传递。

9.6.3　打包后 Log4j 支持问题

打包成.war 部署到 WebLogic 后，会出现如下问题。

示例 9-7：

```
Error: weblogic.management.DeploymentException: Cannot set web app root
system property when WAR file is not expanded - with nested exception:
[java.lang.IllegalStateException: Cannot set web app root system property
when WAR file is not expanded]
```

问题解决：通常您不需要亲自编写 Servlet 或者 Listener，比如直接利用 Log4j 的 com.apache.jakarta.log4j.Log4jInit 类，Spring 的 org.springframework.web.util.Log4jConfig-Servlet 和 org.springframework.web.util.ServletContextListener 方式配置，找到 Log4jConfig-Servlet 和 ServletContextListener 的源码，它们都在适当的地方(callback method)调用了 Log4jWebConfigurer.initLogging(getServletContext())，定位到这个方法，第一句就是 WebUtils.setWebAppRootSystemProperty(servletContext);再定位到该方法，方法很短。

示例 9-8：

```
public static void setWebAppRootSystemProperty(ServletContext servlet-
    Context) throws IllegalStateException {
    String param = servletContext.getInitParameter(WEB_APP_ROOT_KEY_
    PARAM);
    String key = (param != null ? param : DEFAULT WEB APP ROOT KEY);
    String oldValue = System.getProperty(key);
    if (oldValue != null) {
            throw new IllegalStateException("WARNING: Web app root system
            property already set: " + key + " = " + oldValue + " - Choose unique
            webAppRootKey values in your web.xml files!");
    }
    String root = servletContext.getRealPath("/");
    if (root == null) {
            throw new IllegalStateException("Cannot set web app root system
            property when WAR file is not expanded");
    }
    System.setProperty(key, root);
    servletContext.log("Set web app root system property: " + key + " = "
    + root);
}
```

系统需要读取 webAppRootKey 这个参数，所以在部署到 WebLogic 里的时候，在

web.xml 中手动添加如下代码。

示例 9-9：

```
<context-param>
<param-name>webAppRootKey</param-name>
<param-value>webapp.root</param-value>
</context-param>
```

WebLogic 自身也包含对 Log4j 的支持，在打包部署（.war）的时候，会和 Spring 的 org.springframework.web.util.Log4jConfigListener 有冲突，所以改用 Servlet 加载。

web.xml 中删除下面代码。

示例 9-10：

```
<listener id="log4jConfigListener">
    <listener-class>org.springframework.web.util.Log4jConfigListener
    </listener-class>
</listener>
```

将 Listener 加载改为通过 Servlet 加载，再在 web.xml 增加如下代码。

示例 9-11：

```
<servlet>
    <servlet-name>log4jConfigListener</servlet-name>
    <servlet-class>org.springframework.web.util.Log4jConfigServlet
    </servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>
```

9.6.4 Axis 远程调用.net Web Service 问题

调用时出现如下异常：

java.lang.NoSuchMethodError: javax.xml.namespace.QName.getPrefix()Ljava/lang/String。

应用系统需要调用远程.net 平台的 Web Service 接口，该程序在 Tomcat 和 Windows 下 BEA WebLogic 8.1 SP5 下进行测试，全部正常使用，但移植到 HP-UX 上时，每次调用接口时都会找不到 javax.xml.namespace.QName.getPrefix()方法。

查明该方法存在于 jaxrpc.jar 文件中，而 webservices.jar 存在名为 javax.xml.namespace.QName 的重名类。在 startWebLogic.sh 文件中，手动将 jaxpc.jar 排在 webservices.jar 之前进行加载，即可解决该问题。

第 4 篇

诊　断　篇

第 10 章　如何发现问题

备注：DOMAIN_NAME 是在其中定位域的目录的名称，而 ADMIN_SERVER_NAME 是管理服务器的名称，SERVER_NAME 为被管服务的名称。

10.1　WebLogic 监控

WebLogic 监控的目的：发现系统中的隐患及系统运行是否稳定。主要从以下几方面进行检查。

10.1.1　操作系统检查

检查系统 CPU、内存和 I/O 等使用是否异常。如在负载不大的情况下，CPU 是否一直居高不下，内存占用和 I/O 是否一直很大。

可以通过 topas、vmstat、ps 等命令查看。

10.1.2　网络检查

（1）位于一个 Domain 中的各个服务器是否能够联通。

（2）WebLogic 服务器与数据库服务器的链接是否畅通。

（3）集群环境的多播通信是否正常。

10.1.3　WebLogic 检查

（1）检查是否对 JVM 进行了优化，如最大堆内存、最小堆内存以及 GC 算法是否合理。

（2）检查 GC 是否正常。主要是通过 WebLogic 控制台查看 JVM 的空闲内存的变化情况、每次 GC 的回收情况。特别是可以在控制台强制垃圾回收，看看回收的内存是否太小。如果回收的内存太小，说明可能存在内存溢出的隐患。

还可以在服务的启动脚本中添加 GC 日志，从 GC 日志中查看 GC 的回收情况是否正常。

（3）检查线程数。通过 WebLogic 控制台可以查看线程数的统计信息。WebLogic 9 以上的版本的线程是自优化的。但应该查看系统的线程最大数是否过大，如果超过 100 了，就要注意系统为什么会有这么大的压力了。

（4）线程是否有 stuck 状态。线程 stuck 状态说明存在超时的线程，也有可能存在线程的死锁。在查看线程数的时候，查看 stuck 线程统计一栏是否大于 0。如果 stuck 的线程一直大于 0，就要查看系统的运行状态、JVM 进程的状态和内存使用率是否正常等。如果有 Thread dump 文件，或 Java core 文件，就可以分析服务器线程之间是否存在死锁，以及哪些线程处于 stuck 状态。产生 Thread dump，可以通过命令 kill -3 <wls_pid>来解决。

（5）JDBC 连接池。检查连接池中等待连接的数目是否过大，可以做适当调整。

如果用户访问系统变慢，且连接池基本占满，但是 WebLogic 的线程数量很少，就要怀疑应用是否没有释放数据库连接。

（6）系统日志。通过系统的日志分析 WebLogic 服务器及应用程序出现的错误，找到可能影响系统性能的服务器和应用的地方，举例如下。

1．已经对用户进行响应

示例 10-1：

```
java.lang.IllegalStateException: Cannot forward a response that is already
committed
at weblogic.servlet.internal.RequestDispatcherImpl.forward
(RequestDispatcherImpl.java:110)
at com.qtong.hebmc.ExceptionFilter.doFilter(ExceptionFilter.java:55)
```

如上异常，如果对用户的 response 已经提交给用户了，就要怀疑代码中某些地方在过滤器过滤之前已经显示的提交或关闭了对用户响应的 outputstream。有些时候在过滤器中要做后续的一些处理，这时候 filter 中出现异常，有可能导致数据库连接不释放及一些后续的处理得不到执行。

2．线程一直处于 stuck 状态

示例 10-2：

```
<[STUCK] ExecuteThread: '32' for queue: 'weblogic.kernel.Default
(self-tuning)' has been busy for "632" seconds working on the request "Http
Request: /stat/test/TestCtrl.statOracleTest.do", which is more than the
configured time (StuckThreadMaxTime) of "600" seconds.
```

这个示例表明线程 32 执行时间超过 600 秒设定的时间。检查这个请求是否会导致线程执行超时，是否是长时间的处理。当前系统状态是否正常，是否是 CPU、内存等资源不足导致的线程执行缓慢。

重要的一点要查看线程 32 是否变成 unstuck 状态。否则就有可能出现线程死锁。

示例 10-3：

```
<[STUCK] ExecuteThread: '32' for queue: 'weblogic.kernel.Default (self-
tuning)' has become "unstuck".>
```

3．数据库连接问题

示例 10-4：

```
java.sql.SQLException: [BEA][Oracle JDBC Driver]Error establishing socket
to host and port: landingbj:1521. Reason: Connection refused
```

当出现此问题时，一般系统都执行缓慢。检查是否 ping 通数据库，是否有权限，也可以请数据库工程师查看数据库是否运行正常。检查 DNS 服务器或本地 DNS 映射是否正常及域名 dbnew2_vip 能否解析。

4．检查 WebLogic 记录的 Error 级别的日志。

找到可能影响系统运行及功能的问题。

示例 10-5：

```
####<Feb 28, 2010 4:26:20 PM GMT+08:00> <Error> <HTTP> <landingbj> <wls6>
<ExecuteThread: '1' for queue: 'weblogic.socket.Muxer'> <<WLS Kernel>> <>
<> <1267345580932> <BEA-101215> <Malformed Request "/service/js/mareal/
mareal.landingbj</option><option value=". Request parsing failed, Code: -1>
```

此处，说明线程 1 执行请求"/service/js/mareal/mareal. landingbj </option><option value="的时候，WebLogic 对这个请求解析失败，该请求的 URL 明显很不正常，可以请工程师修改对应的应用代码，以免影响应用的功能。

（1）class 文件找不到。

示例 10-6：

```
java.lang.ClassNotFoundException: com.landingbj.controller.js.mareal.
mareal.jobile.com.service.LoginCtrl
at java.lang.Class.forNameImpl(Native Method)
at java.lang.Class.forName(Class.java:130)
at com.mareal.sterna.Controller.parseController(Controller.java:67)
... 20 more
```

检查控制器是否书写正确，还是这个 Java 文件没有正确编译部署。

（2）邮件服务器异常

示例 10-7：

```
org.apache.commons.mail.EmailException: Sending the email to the following
server failed :landingbj:25
Caused by:
javax.mail.MessagingException: Could not connect to SMTP host:landingbj,
port: 25;
  nested exception is:
java.net.SocketException: Connection timed out:could be due to invalid
address
```

检查邮件服务器是否有问题，IP 是否冲突，网络连接是否正常，以免影响应用的功能。

（3）是否是算法的 bug 导致的 OOM。

示例 10-8：

```
java.lang.OutOfMemoryError: Initializing Writer
at com.sun.imageio.plugins.jpeg.JPEGImageWriter.initJPEGImageWriter
(Native Method)
at com.sun.imageio.plugins.jpeg.JPEGImageWriter.<init>(JPEGImageWriter.
java:206)
at com.sun.imageio.plugins.jpeg.JPEGImageWriterSpi.createWriterInstance
(JPEGImageWriterSpi.java:130)
at javax.imageio.spi.ImageWriterSpi.createWriterInstance(ImageWriterSpi.
java:358)
at javax.imageio.ImageIO$ImageWriterIterator.next(ImageIO.java:851)
at javax.imageio.ImageIO$ImageWriterIterator.next(ImageIO.java:835)
at javax.imageio.ImageIO.write(ImageIO.java:1473)
at javax.imageio.ImageIO.write(ImageIO.java:1554)
at com.landingbj.controller.LoginCtrl.smsRandomPic(LoginCtrl.java:267)
... 21 more
```

应该查看 com.sun.imageio.plugins.jpeg.JPEGImageWriter 是否存在 bug，处理大图片时，是否会导致 OOM。

（4）字符越界异常。

示例 10-9：

```
java.lang.StringIndexOutOfBoundsException
at java.lang.String.substring(String.java:1088)
at com.landingbj.controller.nethall.ChongZhiKaChaXunCtrl.transact
(ChongZhiKaChaXunCtrl.java:25)
```

10.2　日志文件的获取

WebLogic 主要的日志文件有如下几种。

10.2.1　server_name.log

该日志记录的是服务（包括 Admin Server 和 Managed Server）启动过程中和关闭过程中的日志，还包括部署在服务上面的应用运行过程中产生的日志。

Server_name.log 的路径为 DOMAIN_NAME/servers/SERVER_NAME/logs/server_name.log。

10.2.2 access.log

该文件具体记录在某个时间，某个 IP 地址的客户端访问了服务器上的那个文件。

Access.log 文件的路径为 DOMAIN_NAME/servers/SERVER_NAME/logs/access.log。

10.2.3 GC.log

记录服务的内存垃圾回收情况。该日志默认情况下并不生成，如果需要生成垃圾回收日志，需要在启动服务的脚本中指定垃圾回收日志文件的路径和名称，也可以在域目录下的 setDomainEnv.sh 或者 wls_home 下的 commEnv.sh 文件中指定。

10.2.4 domain_name.log

记录一个 Domain 的运行情况，一个 Domain 中的各个 WebLogic Server 可以把它们的一些运行信息（比如：很严重的错误）发送给一个 Domain 的 ADMINISTRATOR SERVER 上，ADMINISTRATOR SERVER 把这些信息写到 Domain 日志中。

域日志文件的默认名称和位置是 DOMAIN_NAME/servers/ADMIN_SERVER_NAME/logs/DOMAIN_NAME.log。

10.2.5 jms.messages.log

JMS 服务器日志文件包含有关基本消息生命周期事件的信息，例如消息生成、使用和删除。如果承载主题消息的 JMS 目标配置为启用消息日志记录，则每个基本消息生命周期事件都会在 JMS 消息日志文件中生成一个消息日志事件。

消息日志位于服务器实例根目录下的 logs 目录中，即 DOMAIN_NAME/servers/SERVER_NAME/logs/jmsServers/SERVER_NAMEJMSServer/jms.messages.log。

10.3 启动脚本与配置参数文件的获取

WebLogic 的 Admin Server 的启动脚本为 DOMAIN_NAME/bin/目录下的 startWebLogic.sh，Managed Server 的启动脚本为 DOMAIN_NAME/bin/ startManagedWebLogic.sh。

WebLogic 的参数配置文件有两个，一个是域下的参数配置文件 DOMAIN_NAME/bin/setDomainEnv.sh，另外一个是 WLS_HOME/common/bin/ commEnv.sh。这两个文件的区别是 commEnv.sh 的作用范围是整个 WebLogic 下的所有域，setDomainEnv.sh 的作用范围仅仅是所在的 Domain，如果两个文件中定义的参数有重复，则启动脚本以 setDomainEnv.sh 中的参数为准。

WebLogic 自带的启动脚本不但目录比较深，启动被管服务器的时候，还得在启动脚本

后面加参数，而且还不能在后台运行。 一般在生产环境中，维护人员会重新写一个 WebLogic 的启动脚本，放在一个容易找到的目录中，下面是一个启动 Managed Server 脚本的例子。

示例 10-10：

```
 #!/bin/sh
USER_MEM_ARGS="-Xms4096m -Xmx4096m"
export USER_MEM_ARGS
#======================
nohup /home/weblogic/bea/user_projects/domains/landingbj/bin/start-
ManagedWebLogic.sh App161 8001 http://127.0.0.1:7001 >> /home/weblogic/
bea/user_projects/domains/landingbj/bin/landingbj.out &
```

10.4　Thread dump 的获取和分析

10.4.1　什么是 Thread dump

每启动一个 WebLogic 实例，在系统中就会相应地产生一个 Java 进程。Thread dump 就是把 Java 进程中所有线程的运行状况输出到指定的文件中。

10.4.2　如何获取 Thread dump

Thread dump 的获取在 Windows 下和类 UNIX 系统上的方法并不相同。在 Windows 下，单击激活运行 WebLogic 服务的命令行窗口，然后按 Ctrl +Break 键，即可在屏幕输出中获取 Thread dump。在类 UNIX 系统中，solaris 系统执行 kill-quit WLS 进程号，其他执行 kill -3 WLS 进程号，即可生成 Thread dump。

Thread dump 生成后，还会输出到启动 WebLogic 服务的时候指定的重定向文件中，如下面的启动脚本。

示例 10-11：

```
nohup /home/weblogic/bea/user_projects/domains/landingbj/bin/start-
ManagedWebLogic.sh App161_8001 http://127.0.0.1:7001 >> /home/weblogic/
log/landingbj.out &
```

Thread dump 生成后就可以在/home/weblogic/log/ landingbj.out 文件中获取。

10.4.3　Thread dump 分析说明

Thread dump 生成结果中包含各个线程的线程的运行状态、标识和调用的堆栈；调用的堆栈包含完整的类名，所执行的方法，如果可能的话还有源代码的行数。这些信息可以

帮助我们分析 WebLogic 的运行状况。

WebLogic 中的线程主要状态有 3 种，正在运行的线程、空闲线程和等锁的线程，以下举例说明。

1. WebLogic 中的空闲线程（线程主动 wait）

示例 10-12：

```
"[ACTIVE] ExecuteThread: '30' for queue: 'weblogic.kernel.Default
(self-tuning)'" daemon prio=1 tid=0x0000000040ae9d50 nid=0x6ccf in
Object.wait() [0x00007f6e6d48f000..0x00007f6e6d48fd90]
    at java.lang.Object.wait(Native Method)
    - waiting on <0x00007f6eab8764d8> (a weblogic.work.ExecuteThread)
    at java.lang.Object.wait(Object.java:474)
    at weblogic.work.ExecuteThread.waitForRequest(ExecuteThread.java: 165)
    - locked <0x00007f6eab8764d8> (a weblogic.work.ExecuteThread)
    at weblogic.work.ExecuteThread.run(ExecuteThread.java:186)
```

该线程是一个空闲的线程，weblogic.work.ExecuteThread.waitForRequest 表示该线程在等待请求。

2. 正在执行的线程

示例 10-13：

```
"[ACTIVE] ExecuteThread: '219' for queue: 'weblogic.kernel.Default
 (self-tuning)'" daemon prio=1 tid=0x00007f6e50345a50 nid=0x6e16 runnable
 [0x00007f6e5d668000..0x00007f6e5d66bb10]
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:129)
    at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
    at java.io.BufferedInputStream.read(BufferedInputStream.java:235)
    - locked <0x00007f6f40cc0220> (a java.io.BufferedInputStream)
    at weblogic.net.http.MessageHeader.isHTTP(MessageHeader.java:220)
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
```

runnable 表示该线程正在运行，从 at java.net.SocketInputStream.socketRead0（Native Method）以下表示该线程调用的堆栈。

3. 等锁的线程

示例 10-14：

```
"[ACTIVE] ExecuteThread: '215' for queue: 'weblogic.kernel.Default
(self-tuning)'" daemon prio=1 tid=0x00007f6e50bd84d0 nid=0x6e12 waiting for
```

```
monitor entry [0x00007f6e5d72d000..0x00007f6e5d72fd10]
     at weblogic.utils.classloaders.ChangeAwareClassLoader.loadClass
     (ChangeAwareClassLoader.java:35)
     - waiting to lock <0x00007f6eacbca018> (a weblogic.utils.classloaders.
     ChangeAwareClassLoader)
     at javax.xml.parsers.FactoryFinder.newInstance(FactoryFinder.java:88)
     at javax.xml.parsers.FactoryFinder.findJarServiceProvider(Factory-
     Finder.java:278)
     at javax.xml.parsers.FactoryFinder.find(FactoryFinder.java:185)
     at javax.xml.parsers.DocumentBuilderFactory.newInstance
     (DocumentBuilderFactory.java:98)
………………………………………………………………………………..
………………………………………………………………………………..
………………………………………………………………………………..
```

该线程（waiting for monitor entry）在等待某一个锁（waiting to lock <0x00007f6eacbca018>）时，该锁被其他线程释放后，线程会重新 runnable。

在实际的 Thread dump 分析中，我们主要关注的是正在运行的线程和等锁的线程。

10.4.4　实际环境中 Thread dump 分析示例

以下是生产环境中的一个例子，该线程执行超时，从堆栈信息 net/SocketNativeIO 和 com/informix/jdbc/IfxSqli.executeStatementQuery 看，该线程在通过网络获取数据库的数据。执行数据库查询的代码段应如下。

示例 10-15：

```
"com/xxxxx/prpall/service/spring/PrpWorkbenchMainServiceSpringImpl.getP
rpCFlowInfos"
[STUCK] ExecuteThread: '136' for queue: 'weblogic.kernel.Default (self-
tuning)'" id=150 idx=0x2b0 tid=2639 prio=1 alive, in native, daemon
    at jrockit/net/SocketNativeIO.readBytesPinned(Ljava/io/
    FileDescriptor;[BIII)I(Native Method)
    at jrockit/net/SocketNativeIO.socketRead(SocketNativeIO.java:46)
    at java/net/SocketInputStream.socketRead0(Ljava/io/FileDescriptor;
    [BIII)I(SocketInputStream.java)
    at java/net/SocketInputStream.read(SocketInputStream.java:129)
    at java/io/BufferedInputStream.fill(BufferedInputStream.java:218)
    at java/io/BufferedInputStream.read1(BufferedInputStream.java:256)
    at java/io/BufferedInputStream.read(BufferedInputStream.java:313)
    ^-- Holding lock: java/io/BufferedInputStream@0x7fe572257740[thin lock]
    at com/informix/asf/IfxDataInputStream.readFully(IfxDataInputStream.
    java:146)
    at com/informix/asf/IfxDataInputStream.readSmallInt(IfxDataInput-
    Stream.java:453)
```

```
at com/informix/jdbc/IfxSqli.receiveMessage(IfxSqli.java:2495)
at com/informix/jdbc/IfxSqli.a(IfxSqli.java:1752)
at com/informix/jdbc/IfxSqli.executeStatementQuery(IfxSqli.java:1704)
at com/informix/jdbc/IfxSqli.executeStatementQuery(IfxSqli.java:1635)
at com/informix/jdbc/IfxResultSet.a(IfxResultSet.java:206)
at com/informix/jdbc/IfxStatement.executeQueryImpl(IfxStatement.java:
1229)
at com/informix/jdbc/IfxPreparedStatement.executeQuery(IfxPrepared-
Statement.java:376)
at weblogic/jdbc/wrapper/PreparedStatement.executeQuery(Prepared-
Statement.java:100)
at org/hibernate/jdbc/AbstractBatcher.getResultSet(AbstractBatcher.
java:208)
at org/hibernate/loader/Loader.getResultSet(Loader.java:1812)
at org/hibernate/loader/Loader.doQuery(Loader.java:697)
at org/hibernate/loader/Loader.doQueryAndInitializeNonLazyCollections
 (Loader.java:259)
at org/hibernate/loader/Loader.doList(Loader.java:2232)
at org/hibernate/loader/Loader.listIgnoreQueryCache(Loader.java:2129)
at org/hibernate/loader/Loader.list(Loader.java:2124)
at org/hibernate/loader/hql/QueryLoader.list(QueryLoader.java:401)
at org/hibernate/hql/ast/QueryTranslatorImpl.list(QueryTranslator-
Impl.java:363)
at org/hibernate/engine/query/HQLQueryPlan.performList(HQLQueryPlan.
java:196)
at org/hibernate/impl/SessionImpl.list(SessionImpl.java:1149)
at org/hibernate/impl/QueryImpl.list(QueryImpl.java:102)
at ins/framework/dao/EntityDaoHibernate$5.doInHibernate(EntityDao-
Hibernate.java:506)
at org/springframework/orm/hibernate3/HibernateTemplate.doExecute
(HibernateTemplate.java:419)
at org/springframework/orm/hibernate3/HibernateTemplate.executeFind
(HibernateTemplate.java:343)
at ins/framework/dao/EntityDaoHibernate.findByHql(EntityDaoHibernate.
java:492)
at com/xxxxx/prpall/service/spring/PrpWorkbenchMainServiceSpringImpl.
getPrpCFlowInfos(PrpWorkbenchMainServiceSpringImpl.java:194)
at com/xxxxx/prpall/service/spring/PrpWorkbenchMainServiceSpring-
Impl$$FastClassByCGLIB$$6bb9d24b.invoke(ILjava/lang/Object;[Ljava/
lang/Object;)Ljava/lang/Object;(<generated>)
at net/sf/cglib/proxy/MethodProxy.invoke(MethodProxy.java:149)
at org/springframework/aop/framework/Cglib2AopProxy$CglibMethod-
Invocation.invokeJoinpoint(Cglib2AopProxy.java:700)
at org/springframework/aop/framework/ReflectiveMethodInvocation.
```

```
proceed(ReflectiveMethodInvocation.java:149)
at ins/framework/log/RunTimeLogAdvice.invoke(RunTimeLogAdvice.java:31)
at org/springframework/aop/framework/ReflectiveMethodInvocation.
proceed(ReflectiveMethodInvocation.java:171)
at org/springframework/transaction/interceptor/TransactionInterceptor.
invoke(TransactionInterceptor.java:106)
at org/springframework/aop/framework/ReflectiveMethodInvocation.
proceed(ReflectiveMethodInvocation.java:171)
at org/springframework/aop/interceptor/ExposeInvocationInterceptor.
invoke(ExposeInvocationInterceptor.java:89)
at org/springframework/aop/framework/ReflectiveMethodInvocation.
proceed(ReflectiveMethodInvocation.java:171)
at org/springframework/aop/framework/Cglib2AopProxy$DynamicAdvised-
Interceptor.intercept(Cglib2AopProxy.java:635)
at com/xxxxx/prpall/service/spring/PrpWorkbenchMainServiceSpringImpl
$$EnhancerByCGLIB$$b7d0b209.getPrpCFlowInfos(Ljava/lang/String;
Ljava/lang/String;Ljava/lang/String;II)Lins/framework/common/Page;
(<generated>)
at com/xxxxx/prpall/web/PrpWorkbenchMainAction.findTodoList(Prp-
WorkbenchMainAction.java:120)
at com/xxxxx/prpall/web/PrpWorkbenchMainAction$$FastClassByCGLIB$$27-
bc1495.invoke(ILjava/lang/Object;[Ljava/lang/Object;)Ljava/lang/
Object;(<generated>)
at net/sf/cglib/proxy/MethodProxy.invoke(MethodProxy.java:149)
at org/springframework/aop/framework/Cglib2AopProxy$CglibMethod-
Invocation.invokeJoinpoint(Cglib2AopProxy.java:700)
at org/springframework/aop/framework/ReflectiveMethodInvocation.
proceed(ReflectiveMethodInvocation.java:149)
at ins/framework/log/RunTimeLogAdvice.invoke(RunTimeLogAdvice.
java:31)
at org/springframework/aop/framework/ReflectiveMethodInvocation.
proceed(ReflectiveMethodInvocation.java:171)
at org/springframework/aop/interceptor/ExposeInvocationInterceptor.
invoke(ExposeInvocationInterceptor.java:89)
at org/springframework/aop/framework/ReflectiveMethodInvocation.
proceed(ReflectiveMethodInvocation.java:171)
at org/springframework/aop/framework/Cglib2AopProxy$DynamicAdvised-
Interceptor.intercept(Cglib2AopProxy.java:635)
at com/xxxxx/prpall/web/PrpWorkbenchMainAction$$EnhancerByCGLIB$$
d8e9d0fa.findTodoList()Ljava/lang/String;(<generated>)
at sun/reflect/GeneratedMethodAccessor5830.invoke(Ljava/lang/Object;
 [Ljava/lang/Object;)Ljava/lang/Object;(Unknown Source)
at sun/reflect/DelegatingMethodAccessorImpl.invoke(DelegatingMethod-
AccessorImpl.java:25)
```

```
at java/lang/reflect/Method.invoke(Method.java:585)
at com/opensymphony/xwork2/DefaultActionInvocation.invokeAction-
(DefaultActionInvocation.java:440)
at com/opensymphony/xwork2/DefaultActionInvocation.invokeActionOnly
(DefaultActionInvocation.java:279)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke
(DefaultActionInvocation.java:242)
at com/opensymphony/xwork2/interceptor/DefaultWorkflowInterceptor.
doIntercept(DefaultWorkflowInterceptor.java:163)
at com/opensymphony/xwork2/interceptor/MethodFilterInterceptor.
intercept(MethodFilterInterceptor.java:87)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/ConversionErrorInterceptor.
intercept(ConversionErrorInterceptor.java:122)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/ParametersInterceptor.
doIntercept(ParametersInterceptor.java:195)
at com/opensymphony/xwork2/interceptor/MethodFilterInterceptor.
intercept(MethodFilterInterceptor.java:87)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/ParametersInterceptor.
doIntercept(ParametersInterceptor.java:195)
at com/opensymphony/xwork2/interceptor/MethodFilterInterceptor.
intercept(MethodFilterInterceptor.java:87)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/StaticParametersInterceptor.
intercept(StaticParametersInterceptor.java:148)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at org/apache/struts2/interceptor/CheckboxInterceptor.intercept
(CheckboxInterceptor.java:93)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at org/apache/struts2/interceptor/FileUploadInterceptor.intercept
(FileUploadInterceptor.java:235)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/ModelDrivenInterceptor.
intercept(ModelDrivenInterceptor.java:89)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
```

```
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/ScopedModelDrivenInterceptor.
intercept(ScopedModelDrivenInterceptor.java:128)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at org/apache/struts2/interceptor/ProfilingActivationInterceptor.-
intercept(ProfilingActivationInterceptor.java:104)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at org/apache/struts2/interceptor/debugging/DebuggingInterceptor.
intercept(DebuggingInterceptor.java:267)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/ChainingInterceptor.intercept
(ChainingInterceptor.java:126)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/PrepareInterceptor.doIntercept
(PrepareInterceptor.java:138)
at com/opensymphony/xwork2/interceptor/MethodFilterInterceptor.
intercept(MethodFilterInterceptor.java:87)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/I18nInterceptor.intercept-
(I18nInterceptor.java:148)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at org/apache/struts2/interceptor/ServletConfigInterceptor.intercept
 (ServletConfigInterceptor.java:164)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/AliasInterceptor.intercept-
 (AliasInterceptor.java:128)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at com/opensymphony/xwork2/interceptor/ExceptionMappingInterceptor.
intercept(ExceptionMappingInterceptor.java:176)
at com/opensymphony/xwork2/DefaultActionInvocation.invoke(Default-
ActionInvocation.java:236)
at org/apache/struts2/impl/StrutsActionProxy.execute(StrutsAction-
Proxy.java:52)
at org/apache/struts2/dispatcher/Dispatcher.serviceAction(Dispatcher.
java:468)
at org/apache/struts2/dispatcher/FilterDispatcher.doFilter(Filter-
```

```
Dispatcher.java:395)
at weblogic/servlet/internal/FilterChainImpl.doFilter(FilterChain-
Impl.java:42)
at org/springframework/orm/hibernate3/support/OpenSessionInView-
Filter.doFilterInternal(OpenSessionInViewFilter.java:198)
at org/springframework/web/filter/OncePerRequestFilter.
doFilter(OncePerRequestFilter.java:76)
at weblogic/servlet/internal/FilterChainImpl.doFilter(FilterChain-
Impl.java:42)
at com/xxxxx/prpall/common/util/PowerFilter.doFilter(Power-
Filter.java:60)
at weblogic/servlet/internal/FilterChainImpl.doFilter(Filter-
ChainImpl.java:42)
at com/xxxxx/prpall/common/util/SessionFilter.doFilter(SessionFilter.
java:71)
at weblogic/servlet/internal/FilterChainImpl.doFilter(FilterChain
Impl.java:42)
at org/springframework/web/filter/CharacterEncodingFilter.doFilter-
Internal(CharacterEncodingFilter.java:96)
at org/springframework/web/filter/OncePerRequestFilter.doFilter
(OncePerRequestFilter.java:76)
at weblogic/servlet/internal/FilterChainImpl.doFilter(FilterChain-
Impl.java:42)
at ins/framework/web/CompressFilter.doFilter(CompressFilter.java:87)
at weblogic/servlet/internal/FilterChainImpl.doFilter(FilterChain-
Impl.java:42)
at cn/com/xxxxx/cas/filter/CustomCASFilter.doFilter(CustomCASFilter.
java:253)
at weblogic/servlet/internal/FilterChainImpl.doFilter(FilterChain-
Impl.java:42)
at weblogic/servlet/internal/WebAppServletContext$ServletInvocation-
Action.run(WebAppServletContext.java:3242)
at weblogic/security/acl/internal/AuthenticatedSubject.doAs
(AuthenticatedSubject.java:321)
at weblogic/security/service/SecurityManager.runAs(SecurityManager.
java:121)
at weblogic/servlet/internal/WebAppServletContext.securedExecute
(WebAppServletContext.java:2010)
at weblogic/servlet/internal/WebAppServletContext.execute(WebApp-
ServletContext.java:1916)
at weblogic/servlet/internal/ServletRequestImpl.run(Servlet-
RequestImpl.java:1366)
at weblogic/work/ExecuteThread.execute(ExecuteThread.java:209)
at weblogic/work/ExecuteThread.run(ExecuteThread.java:181)
at jrockit/vm/RNI.c2java(JJJJJ)V(Native Method)
```

10.5　Heap dump 的获取和分析

10.5.1　什么是 Heap dump

Java Heap 是所有类实例和数组对象分配的一个运行时数据区，其间所有 Java VM 线程在执行期间共享 Heap 中的数据。那么一个 Java Heap dump 相当于在一个特殊的时间点上生成的一个快照，它就像给一个繁忙的数据库在给定的时间上来了一个照片，我们通过这张快照可以识别哪些组件在哪张快照的哪个时间点上是可用的。

10.5.2　如何获取 Java Heap dump

当 JVM 中对象过多，Java 堆内存（Heap Memory）耗尽时，就可能会触发产生 Heap dump 文件。另外，可以使用工具或命令显式地产生该文件。在命令行中程序执行过程中按 Ctrl+Break 键可以产生，使用工具如 IBM HeapAnalyzer、Sap Memory Analyzer 以及 eclipse memory analyzer，都可以在指定状态产生 Heap dump 文件。

10.5.3　什么是 Jps 和 Jmap

Jps 用于列出所有 Java 相关线程的 pid 等信息。

示例 10-16：

```
[root@myjrjapp-100 ~]# jps
23178 Jps
20289 Bootstrap
```

其中，20289 Bootstrap 是指系统中运行的 Tomcat 进程号和进程名。

Jmap 是一个可以输出所有内存中对象的工具，甚至可以将 VM 中的 heap 以二进制输出成文本。Jmap-dump:format=b,file=f1 3024 可以将 3024 进程的内存 heap 输出到 f1 文件里。它可以打印出某个 Java 进程（使用 pid）内的所有'对象'的情况（如产生哪些对象，及其数量）。

如 Jmap pid 打印内存使用的摘要信息。

10.5.4　Jmap 的作用

1．jmap -heap pid

查看 Java 堆（heap）的使用情况。

示例 10-17：

```
        using thread-local object allocation.
        Parallel GC with 4 thread(s)//GC 方式

Heap Configuration:    //堆内存初始化配置
MinHeapFreeRatio=40    //对应 JVM 启动参数-XX:MinHeapFreeRatio，设置 JVM 堆最小
                         空闲比率(default 40)
MaxHeapFreeRatio=70    //对应 JVM 启动参数 -XX:MaxHeapFreeRatio,设置 JVM 堆最大
                         空闲比率(default 70)
 MaxHeapSize=512.0MB   //对应 JVM 启动参数-XX:MaxHeapSize=，设置 JVM 堆的最大值
NewSize  = 1.0MB       //对应 JVM 启动参数-XX:NewSize=,设置 JVM 堆的'新生代'的默
                         认大小
MaxNewSize =4095MB     //对应 JVM 启动参数-XX:MaxNewSize=，设置 JVM 堆的'新生代'
                         的最大值
OldSize  = 4.0MB       //对应 JVM 启动参数-XX:OldSize=<value>:，设置 JVM 堆'老
                         生代'的大小
NewRatio  = 8    //对应 JVM 启动参数-XX:NewRatio=:，'新生代'和'老生代'的大小比率
SurvivorRatio = 8      //对应 JVM 启动参数-XX:SurvivorRatio=，设置年轻代中 Eden
                         区与 Survivor 区的大小比值
 PermSize= 16.0MB      //对应 JVM 启动参数-XX:PermSize=<value>:，设置 JVM 堆的
                         '永生代'的初始大小
 MaxPermSize=64.0MB    //对应 JVM 启动参数-XX:MaxPermSize=<value>:,设置 JVM 堆
                         的'永生代'的最大值

         Heap Usage:                                   //堆内存分步
         PS Young Generation
         Eden Space:                                   //Eden 区内存分布
           capacity = 20381696 (19.4375MB)             //Eden 区总容量
           used     = 20370032 (19.426376342773438MB)  //Eden 区已使用
           free     = 11664 (0.0111236572265625MB)     //Eden 区剩余容量
           99.94277218147106% used                     //Eden 区使用比率
        From Space:    //其中一个 Survivor 区的内存分布
            capacity = 8519680 (8.125MB)
            used     = 32768 (0.03125MB)
            free     = 8486912 (8.09375MB)
            0.38461538461538464% used
        To Space:      //另一个 Survivor 区的内存分布
           capacity = 9306112 (8.875MB)
           used     = 0 (0.0MB)
           free     = 9306112 (8.875MB)
           0.0% used
        PS Old Generation                              //当前的 Old 区内存分布
           capacity = 366280704 (349.3125MB)
           used     = 322179848 (307.25464630126953MB)
```

```
            free      = 44100856 (42.05785369873047MB)
            87.95982001825573% used
        PS Perm Generation                          //当前的"永生代"内存分布
            capacity = 32243712 (30.75MB)
            used     = 28918584 (27.57891082763672MB)
            free     = 3325128 (3.1710891723632812MB)
            89.68751488662348% used
```

2. jmap -histo pid

查看堆内存（Histogram）中的对象数量、大小。

示例 10-18：

```
    Num        #instances      #bytes        class name

    序号        实例个数         字节数         类名
    ----------------------------------------------
     1:         3174877        107858256      [C
     2:         3171499        76115976       java.lang.String
     3:         1397884        38122240       [B
     4:         214690         37785440       com.landingbj.book.form.Book
     5:         107345         18892720       com.landingbj.book.form.Book
     6:         65645          13953440       [Ljava.lang.Object;
     7:         59627          7648416        <constMethodKlass>
     8:         291852         7004448        java.util.HashMap$Entry
     9:         107349         6871176        [[B
 Total          9150732        353969416
```

3. jmap - dump　pid

将内存使用的详细情况输出到文件。

示例 10-19：

```
map -dump:format=b,file= m.dat pid
```

用 jhat 命令可以参看 jhat -port 5000 m.dat。

10.5.5　Jmap 的分析方法

分析方法 1：

以上两个命令可以结合起来用。

示例 10-20：

```
[root@myjrjapp-100 ~]# jps
```

```
23178 Jps
20289 Bootstrap


[root@myjrjapp-100 ~]# jmap 20289
Attaching to process ID 20289, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 10.0-b19

using thread-local object allocation.
Parallel GC with 8 thread(s)

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 1073741824 (1024.0MB)
   NewSize          = 1048576 (1.0MB)
   MaxNewSize       = 4294901760 (4095.9375MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 8
   SurvivorRatio    = 8
   PermSize         = 134217728 (128.0MB)
   MaxPermSize      = 268435456 (256.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 118358016 (112.875MB)
   used     = 38070328 (36.30669403076172MB)
   free     = 80287688 (76.56830596923828MB)
   32.165398919833194% used
From Space:
   capacity = 458752 (0.4375MB)
   used     = 155664 (0.1484527587890625MB)
   free     = 303088 (0.2890472412109375MB)
   33.932059151785715% used
To Space:
   capacity = 458752 (0.4375MB)
   used     = 0 (0.0MB)
   free     = 458752 (0.4375MB)
   0.0% used
PS Old Generation
   capacity = 954466304 (910.25MB)
   used     = 72784624 (69.41282653808594MB)
   free     = 881681680 (840.8371734619141MB)
   7.625688166776813% used
```

```
PS Perm Generation
   capacity = 134217728 (128.0MB)
   used     = 38192248 (36.42296600341797MB)
   free     = 96025480 (91.57703399658203MB)
   28.455442190170288% used
[root@myjrjapp-100 ~]#
```

分析方法 2：

使用 Jmap 命令 dump 内存出来。

示例 10-21：

```
jmap -dump:live,format=b,file=heap.bin 8023
```

之后会在当前目录创建一个 heap.bin 文件，大小可能会有好几百 MB 甚至几 GB。可以把此文件进行压缩，然后再传到其他 Windows 机器中进行结果分析。

示例 10-22：

```
 [root@openAS-main ~]# gzip heap.bin
```

在测试机上安装一个分析工具：MemoryAnalyzer-Incubation-0.8.0.20100408-win32.win32.x86，这是 Eclipse MAT，是 SAP 公司贡献的一个工具，可以在 Eclipse 网站进行免费下载。用它打开解压后的 dump 文件，就可以在界面上看到内存分析。

10.6　关于 Java dump 的一些常见问题

（1）为什么发生 JVM Crash 时，JVM 没有自动生成 Java dump 文件？

答：这种情况大多与系统的环境变量或者 JVM 启动参数的设置有关，比如设置了 DISABLE_JAVADUMP=true，IBM_NOSIGHANDLER=true，等等，因此可以首先检查系统设置和 JVM 启动参数。当然也不排除因为一些不确定因素导致 JVM 无法产生 Java dump，虽然这种可能性比较小。

（2）Java Dump 中的很多线程处于 state:CW 和 state:B 状态，它们之间有何区别？

答：两者都处于等待状态。不同的是：

CW - Condition Wait-条件等待，这种等待一般是线程主动等待或者正在进行某种 I/O 操作，而并非等待其他线程释放资源。比如 sleep()、wait()、join()等方法的调用。

B-Blocked-线程被阻塞，与条件等待不同，线程被阻塞一般不是线程主动进行的，而是由于当前线程需要的资源正在被其他线程占用，因此不得不等待资源释放以后才能继续执行，例如，synchronized 代码块。

（3）为什么在 PsList 里看到的线程无法映射到 Java dump 中？

答：由于很多操作系统工具和命令输出的线程的 TID 都是十进制的，映射到 Java dump 时首先要将其转换为十六进制数字，然后再到 Java dump 中查找对应的 native ID。Java dump 中每个线程都有两个 ID，一个是 Java 线程的 TID，另一个是对应操作系统线程的 native ID。

第 11 章　常规服务器挂起故障

11.1　服务器挂起概述

11.1.1　什么是服务器挂起

如果 WLS 实例不再响应用户请求，一般称该实例已挂起，具体表现如下。

（1）向服务器发出的新请求未获接受或被忽略。

（2）现有请求超时或一直没有返回响应。

11.1.2　服务器挂起分类

服务器挂起模式具体可以分为如下几类。

（1）线程占用导致服务器挂起模式。

（2）垃圾回收导致服务器挂起模式。

（3）代码优化中服务器挂起模式。

（4）程序死锁服务器挂起模式。

（5）JDBC 相关的服务器挂起模式。

（6）EJB RMI 服务器挂起模式。

（7）JSP 编译导致服务器挂起模式。

（8）JSP 导致服务器挂起模式。

（9）Sun JVM bug 导致服务器挂起模式。

11.1.3　服务器挂起症状

故障症状包括以下内容。

（1）请求未得到处理。

（2）服务器表现为不执行任何操作。

（3）如果服务器正接近挂起，它处理请求的时间会越来越长。

（4）挂起对服务器的影响。

① 可能会因挂起而崩溃，但这不是必然结果。

② 可能会从挂起状态中恢复过来。

例如，如果挂起是由资源争用引起的，则当存在空闲资源时，服务器就可以恢复正常了。

③ 如果不手动执行某种操作，服务器可能会一直保持挂起状态。

11.2　常规服务器挂起故障

11.2.1　服务器挂起成因总述

（1）服务器挂起通常由缺乏某种资源而引起的，这种缺乏使服务器无法对请求进行处理。

① 没有足够的线程或内存来执行处理。

② 没有足够的文件句柄可供服务器用于管理打开的文件。

③ 资源争用或死锁也会导致挂起。

（2）如果服务器似乎已经挂起，但又缓慢恢复正常，那么：

① 资源空闲下来，处理继续执行；

② 恢复过程可能漫长的让人无法接受。

（3）有时候服务器看起来好像已经挂起了，但是可能是正在进行某个资源消耗比较大的操作，当操作完成以后，服务器可能就会恢复正常了。

11.2.2　服务器挂起具体成因

（1）配置的线程数不足。

① 所有线程都被占用了，没有线程可用于处理新工作。

② 详细信息可以参考“线程占用导致服务器挂起模式”。

（2）垃圾回收（Garbage Collection，GC）花费太多时间。

① GC 会影响服务器的性能，因为在 GC 期间服务器处理会暂停。

② 详细信息可以参考“垃圾回收导致服务器挂起模式”。

（3）JVM 在代码优化期间挂起。

① 可能会导致临时挂起。

② 详细信息可以参考“代码优化中服务器挂起模式”。

（4）应用程序死锁。

① 线程 A 锁定资源 1，然后等待锁定资源 2；线程 B 锁定资源 2，然后等待锁定资源 1。

② 详细信息可以参考“应用程序死锁服务器挂起模式”。

（5）JDBC 死锁。

① 发生数据库死锁。

② 详细信息可以参考“JDBC 中的服务器挂起模式”。

（6）所有线程都在等待对远程 JVM 的 RMI 调用响应。

① 大量远程 JNDI 查找导致线程挂起。

② 详细信息可以参考“EJB RMI 服务器挂起模式”。

（7）JSP 编译。

① 服务器在大量负载情况下可能会挂起。

② 详细信息可以参考“JSP 编译导致服务器挂起模式”。

（8）JSP 的 Servlet 时间设置不当。

① 例如，PageCheckSeconds 的设置不当。

② 详细信息可以参考“JSP 导致服务器挂起模式”。

（9）SUN JVM 错误。

① 例如，轻量型线程库中有错误。

② 详细信息可以参考“Sum JVM 错误导致服务器挂起模式”。

上面提到的这些是最常见的导致服务器挂起的因素。如果服务器挂起了，很可能是由于上面的这些因素之一造成的。另外，垃圾回收操作是一个密集处理操作，可能会造成服务器不响应。在手动进行垃圾回收以前，一定要确认是否确实需要进行垃圾回收操作。当您执行垃圾回收操作时，JVM 经常会检查堆中的每一个存活的对象。

11.3 服务器挂起探查

11.3.1 基本探查步骤

（1）从命令行对服务器执行 Ping 命令。

① 执行以下代码：

```
java weblogic.Admin -url t3://localhost:7001 -username <user> -password
<pass> PING;
```

② 如果服务器能做出响应，则可能是应用程序挂起，而服务器并未挂起。

（2）检查服务器是否正在执行垃圾回收。

① 使用-verbose:gc 参数重新启动服务器。

② 如果服务器似乎已挂起，可检查是否正在运行 GC。

如果服务器正在挂起，首先使用上面提到的命令对 WLS 做 Ping 操作。如果服务器可以响应 Ping，那可能只是应用程序挂起了，而不是服务器挂起了。

如果某个应用程序占用了所有的执行线程，正在执行需要长时间运行的代码，WLS 就会没有能力处理 Ping 请求，直到有执行线程可用才可以。同样，要确认服务器确实挂起了，并且没有做垃圾回收操作。如果想要确认的话，重启 WLS，加上-verbose:gc 参数，并且将标准输出和标准错误输出重定向到一个文件里去，如果服务器停止了响应的话，可以根据输出信息来看它是否正在做垃圾回收操作，或者 WLS 是否真的挂起了。

11.3.2 查看执行线程运行状态

查看执行线程的运行情况，如果一个空闲线程都没有的话，那么很可能会导致服务器

挂起，这样可以查看 default 队列里是否有空闲的执行线程。

登录控制台，然后依次执行 DomainName→Servers→ServerName→Monitoring→General→Monitor All Active Queues→weblogic.kernel.Default 命令，里面的 Current Request 列会显示线程是否正在工作。

如果一个空闲线程也没有的话，很可能就需要为应用程序配置更多数量的线程。可以在管理控制台更改线程的数量，在更改完以后，WLS 会保存在 config.xml 文件里。

11.3.3　创建 Thread dump

1. Thread dump 简介

Thread dump 是在特定时刻对 JVM（服务器）进程中所有活动线程的原样快照，因此运行 WLS 的 Java 进程的 Thread dump 里会记录 WLS 自己的执行线程以及应用程序创建的执行线程正在干什么（丢失的线程除外），然后以文本的形式提供给我们，对于探查许多类型的服务器挂起极具价值，另外对分析其他 WLS 故障也很有参考价值。

因为服务器的某些线程等待和挂起可能会导致服务器挂起，而 Thread dump 里又会记录各个未丢失的线程的状态，所以我们可以借助 Thread dump 对 WLS 挂起进行分析。

可以在运行着的服务器上进行，如果该服务器还能响应 Ping（如果 WLS 无法响应 Ping，那么可能就不能对其做 Thread dump 了），或在服务器崩溃（如果发生）的瞬间进行。

另外，在做 Thread dump 时，最好接连做 3 次以上，每次间隔 5～10 秒钟，以辅助死锁的检测，也可以分析各个 Thread dump 之间线程状态随时间发生的变化。

2. 对运行着的服务器上进行 Thread dump

（1）针对 UNIX/Linux 平台。

执行 kill -3　<WLS_pid>指令，其中 WLS_pid 为 WLS 进程的进程号（Pid）。

（2）针对 Windows 平台。

① 进入服务器的“命令提示符”窗口。

② 右键单击标题栏，然后执行“属性”命令。

③ 在“布局”选项卡的“屏幕缓冲区大小”列表框下，将“高度”设置为 2000 以上。

④ 单击“确定”按钮。

⑤ 要将 Thread dump 输出到命令窗口，按 Ctrl+Break 键。

⑥ 在输出中向前回滚到转储的起始处，即以下列词语开头的地方：Full thread dump。

（3）针对各平台通用的命令。

在 Windows/UNIX/Linux 操作系统上都可以执行下面的命令来做 Thread dump：

```
java weblogic.Admin -url ManagedHost:Port -username weblogic -password
weblogic THREAD_DUMP
```



> **注 意**
> 使用命令时也需要 WLS 能够响应 Ping 操作。

3．服务器发生故障时进行 Thread dump

捕捉服务器发生故障前服务器线程的状态。

（1）服务器启动时启用下列项：

```
Sun JVM -xx:+ShowMessageBoxOnError
JRockit JVM  -Djrockit.waitonerror
```

（2）同时在前台启动服务器。

（3）之后如果 JVM 崩溃，则会显示以下提示：

```
Do  you  want  to  dubug  the  problem ?
```

（4）回答提示前，您就可以趁机捕捉 JVM 的 Thread dump 了。

> **注 意**
> 如果 WLS 是由于自己 trap 的错误而被强制关闭的，那么 WLS 就不会提示您进行 Debug。这是因为 WLS 不是技术上的崩溃，而是由于致命错误而“优雅地”自行关闭了。

11.3.4　初始探查结果分析

如果不能对挂起的服务器做前边提到的操作，也就是说无法从挂起的服务器中获得任何信息的话，您可能需要重新启动一下 WLS，在重启时要加适当的参数，以确保在下次出现挂起时可以获得某些数据，其中确保能做 Thread dump 是在做分析时最重要的一步，并且如果可能的话对日志进行记录，以查看在出现挂起时，是否正在进行垃圾回收操作。

如果通过分析发现在服务器挂起时 WLS 正在进行垃圾回收操作，那么您的垃圾回收模式或者设置可能是错误的，可能需要进行调整。比如在业务执行的关键时刻进行垃圾回收。由垃圾回收造成的服务器挂起模式可以参考“垃圾回收服务器挂起模式”，里面有详细的论述。

如果垃圾回收不是原因的话，下一步就要分析 Thread dump，看一下每个线程都在做什么，然后根据每个线程当时的运行情况，再做进一步的分析。比如，如果在服务器挂起时，有很多的线程都在做 JSP 的编译工作，那么就有可能是 JSP 的编译造成的挂起，下一步就要对 JSP 编译做具体的分析了，此时可以参考“JSP 编译导致服务器挂起模式”。以此类推，如果是线程正在做其他工作，那么再针对具体的情况，再做进一步的分析即可。

接下来针对几种具体的情况，做进一步的分析。

1．Socket Reader 线程引起的服务器挂起

如果服务器在挂起时有空闲线程，那么 Socket Reader 线程数不足可能是问题所在。

（1）Socket Reader 线程简介。

Socket Reader 线程实际是执行线程，只是专门拿出执行线程来做接受来自监听线程队列的传入请求，然后将该请求置于服务器的执行线程队列中的工作。

分配执行线程作为 Socket Reader 线程能提高 WLS 接受客户端请求的速度和能力。

默认情况下，WLS 实例在启动时会创建 3 个 Socket Reader 线程，一个线程通常用于执行轮询功能，另外两个线程用于处理请求。具体线程数量是默认执行队列线程数的某个百分比。一般，Socket Reader 线程的数量应该比较小。但如果需要的话，就要增加该线程数。

集群系统需要的 Socket Reader 线程数可能要多于默认的 Socket Reader 线程数，比如在业务高峰阶段使用了 3 个以上的 Socket。

如果服务器充当正在挂起的服务器实例的客户端，需要为每个服务器配置一个线程。

ThreadPoolPercentSocketReaders 属性控制 Socket Reader 线程的数量，它设置从执行线程中拿出来做 Socket Reader 线程的个数的最大百分比。这个数的最优数值与应用程序有关系，默认是 33，也就是拿出执行线程的 33%来做 Socket Reader 线程，这个值的合法范围是 1～99。可以通过设置该属性来增大 Socket Reader 线程的数量。

需要注意的是，专门用来做 Socket Reader 线程的数量和具体执行任务的执行线程之间的数量要达到一个平衡，否则的话 WLS 可能会出现故障。

另外，如果在 Thread dump 里没有 Socket Reader 线程的话，那么 WLS 系统可能在某个地方存在 bug，以至于让 Socket Reader 线程消失了，这时就需要联系 WLS 的技术支持。

（2）Socket Reader Thread dump 示例。

Thread dump 中的 Socket Reader 线程示例如下。

示例 11-1：

```
"ExecuteThread: '2' for queue: 'weblogic.socket.Muxer'" daemon prio=10
tid=0x000 36128 nid=75 lwp_id=6888070 waiting for monitor entry
[0x1b12f000..0x1b12f530]
 at weblogic.socket.PosixSocketMuxer.processSockets
 (PosixSocketMuxer.java:92) - waiting to lock <0x25c01198> (a java.lang.
String)
 at weblogic.socket.SocketReaderRequest.execute (SocketReaderRequest.
java:32). . .
"ExecuteThread: '1' for queue: 'weblogic.socket.Muxer'" daemon prio=10
tid=0x000 35fc8 nid=74 lwp_id=6888067 runnable [0x1b1b0000..0x1b1b0530] at
weblogic.socket.PosixSocketMuxer.poll(Native Method)
 at weblogic.socket.PosixSocketMuxer.processSockets(PosixSocket-
Muxer.java:99)
- locked <0x25c01198> (a java.lang.String)
 at weblogic.socket.SocketReaderRequest.execute(SocketReaderRequest.
java:32). . .
"ExecuteThread: '0' for queue: 'weblogic.socket.Muxer'" daemon prio=10
tid=0x000 35e68 nid=73 lwp_id=6888066 waiting for monitor entry
[0x1b231000..0x1b231530]
```

```
 at weblogic.socket.PosixSocketMuxer.processSockets(PosixSocketMuxer.
java:92)
 - waiting to lock <0x25c01198> (a java.lang.String)
 at weblogic.socket.SocketReaderRequest.execute(SocketReaderRequest.
 java:32). . .
```

以上信息为 Thread dump 中的 Socket Reader 线程示例。其中线程 1 在执行 poll 轮询功能，线程 0 和线程 2 在处理请求，可以从每个线程信息的第三行处获得。线程 1 在调用 poll 方法，线程 2 和 0 在调用 processSockets 方法。

（3）检查监听线程。

首先所有请求都是通过监听线程进入服务器的，如果缺少监听进程，则无法接收任何工作，监听线程应当在 socketAccept 方法中。

Thread dump 中的监听线程示例如下。

示例 11-2：

```
"ListenThread.Default" prio=10 tid=0x00037888 nid=93 lwp id=6888343
runnable [0x 1a81b000..0x1a81b530]
at java.net.PlainSocketImpl.socketAccept(Native Method)
at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
- locked <0x26d9d490> (a java.net.PlainSocketImpl)
at java.net.ServerSocket.implAccept(ServerSocket.java:439)
at java.net.ServerSocket.accept(ServerSocket.java:410)
at weblogic.socket.WeblogicServerSocket.accept (WeblogicServerSocket.
java:24)
at weblogic.t3.srvr.ListenThread.accept(ListenThread.java:713)
at weblogic.t3.srvr.ListenThread.run(ListenThread.java:290)
```

所有请求都是经过监听器线程进入 WLS 系统的。如果监听线程没有了，就无法接收请求，这样也就不能做任何工作了。检查 Thread dump 里确实有监听线程，而且监听线程应该在执行 socketAccept 方法。

2．无空闲线程时的服务器挂起

（1）无空闲线程挂起症状分析。

① 如果通过分析，发现等待、死锁或其他资源限制未表现为一致的模式，但服务器挂起仍定期出现，如果此时检查 Thread dump 信息，发现除了所有繁忙线程外，看不到其他的一致性格式，那么很可能是没有为应用程序分配足够多的线程来做相应的工作。

② 对于高占用率应用程序（如 Web 应用程序），可能需要为其分配更多执行队列（和线程），专供它使用，对于这种情况的具体信息可以“参考线程占用挂起模式”。

③ WebLogic Server 可能检测到卡滞线程，如果服务器检测到卡滞线程，会进入 CRITICAL 状态。及早检测到卡滞状态可以为及时探查（例如，在整个服务器挂起前进行 Thread dump）创造条件。

（2）进行相应调整。

① 针对上述症状的前两种情况，如果通过分析，发现确实是线程不够了，那么就需

要增加执行线程的数量。

a．增加 ThreadCount，以便服务器可以同时执行更多的操作。可以通过 config.xml 文件里执行队列元素的 ThreadCount 属性的值来增大执行线程的数量。

注：执行线程的数量不是增大得越多越好，因为执行线程也会消耗系统资源，如果增大的执行线程数超过必需的线程数以后，可能反而会造成系统性能的下降，并且可能导致 Native 内存不足引起的 OutOfMemory 异常。

b. 设置 ThreadPoolPercentSocketReaders 来定义必需的 SocketReader 线程数(%)；Socket Reader 线程及如何增大其数量已经有所介绍，这里再补充两点。

首先，为了达到最好的 Socket 性能，BEA 建议您使用本地 Socket Reader 实现，而不是使用纯 Java 实现。然而，假如您必须使用纯 Java Socker Reader 实现的话，您仍然可以提高 Socket 通信的性能，为每个服务器实例和客户端 machine 配置适当数量的执行线程来当作 Socket Reader 线程使用。

其次，确认使用了单独的本地 Socket Reader 多路复用线程，即 NativeIOEnabled=true（缺省值）。

3．对于上述症状第三种情况的卡滞线程问题

设置 StuckThreadMaxTime 和 StuckThreadTimerInterval 来控制服务器检测卡滞线程的方式。

StuckThreadMaxTime 参数设置了线程必须持续工作多长时间以后，WLS 将其诊断为卡住线程，默认是 600 秒。

Stuck Thread Timer Interval 参数指定了 WLS 检测线程的时间间隔，也就是每隔多长时间以后 WLS 去扫描一下进程，看看它们是否持续工作了 StuckThreadMaxTime 参数指定的时间，默认也是 600 秒。

4．线程数足够时的服务器挂起

（1）概述。

如果通过分析发现服务器线程数足够，那么可能会出现下面的这几种情况。

① 连续的 Thread dump 分析，可能显示已经达到资源极限。

a．发生 OutOfMemory 异常。

b．发生 Too Many Open Files。

c．达到 JDBC 连接池限值。

② Thread dump 也可能显示发生应用程序死锁。

③ 应用程序处理可能导致了挂起。

a．EJB RMI 调用。

b．JSP 调用。

④ 也可能存在其他情况。

a．正在进行垃圾回收。

b．代码优化。

c．Sun JVM 错误。

d．JSP 编译。

5．EJB RMI 服务器挂起模式

在 WebLogic JNDI 树中存储集群对象及从中检索集群对象方面的不合理设计会导致该模式，通常表现为 wait 方法中存在许多线程，它们在等待其他集群成员向它们传送工作或数据。发生此模式时，Thread dump 中信息显示如下。

Thread dump 中显示此模式的典型故障症状如下。

示例 11-3：

```
. . .
at weblogic.rjvm.ResponseImpl.waitForData(ResponseImpl.java:72)
. . .
```

6．JDBC 连接池资源耗尽服务器挂起模式

在 JDBC 连接池中的资源耗尽时，服务器可能会因等待可用的资源而挂起。此时 Thread dump 将显示 JDBC 子系统或 JDBC 驱动程序代码中有正在等待连接的线程。应该参照以下所述设置 JDBC 连接池的大小。

- 连接池大小必须大到可以处理所有并发请求。
- 池大小应与执行线程数相等。

关于 JDBC 造成的服务器挂起模式，可以参考“JDBC 中的服务器挂起模式”。

11.4 故障排除检查清单

（1）确认服务器是否已挂起，并确定服务器状态：崩溃、最终恢复正常，还是这两种情况均未发生。

（2）收集初始探查数据如下。

① 如果没有数据，可设置参数以供下次发生挂起时使用。

② 检查线程活动并进行若干次 Thread dump。

（3）查找服务器挂起的根本原因。

① 可用的执行线程数和 Socket Reader 线程数是否充足？是否有监听线程？

② GC 是否正在运行？

③ 如果不属上述情况，可对 Thread dump 进行分析，查找可能的成因，并参考使用的特定模式。

（4）根据需要更改配置，增大导致服务器挂起的限值。

（5）根据需要探查具体的故障成因。

① 参考特定模式。

② 进行更改，以分析或消除可疑的成因。

③ 确保启用了 Thread dump 的记录功能和/或设置，以便进一步捕捉数据。

（6）继续监视，看是否还会发生故障。

11.4.1 垃圾回收导致服务器挂起

1. 什么是垃圾回收导致服务器挂起

GC 分类如下。

（1）Scavenge GC。一般情况下，当新对象生成，并且在 Eden 申请空间失败时，就会触发 Scavenge GC，对 Eden 区域进行 GC，清除非存活的对象，并且把仍存活的对象移动到 Survivor；整理 Survivor 的 From 及 To 区域。

（2）Full GC。导致 Full GC 发生的可能原因有很多，比如以下几方面。

- Old 被写满。
- 某些情况下的 Survivor 被写满或在 Survivor 无法申请存储空间。
- 上一次 GC 之后 Heap 的各区域分配策略动态变化。
- Perm 域被写满。
- System.gc()函数在应用程序中被显式地调用。
- 发生 Full GC 的后果是，Full GC 会对整个 Heap 包括 Eden、Survivor、Old 3 个区域进行 GC，删除所有非存活对象，并且移动存活的对象。

Scavenge GC 在 New 区域中完成，因此速度更快。Full GC 同时包含了 New 和 Old 的区域，因此速度比 Scavenges 要慢。

频繁的 Full GC 导致 CPU 使用过多时，其他线程都等待，整体性能下降，就有可能导致服务器挂起。但这个 Full GC 是由一定的原因导致的。

2. 垃圾回收导致服务器挂起的探查

（1）JVM 配置的不合理。

- New 区不宜太小，否则 Scavange GC 太频繁；但不宜太大，否则每次 GC 时间过长。
- 设定合理的 SurvivorRatio，使得避免 Overflow 的出现。
- 避免 Permanent 区不够用。
- 设定合理的 Heap 区的大小，过大，则 Full GC 时耗时过长，或使用 Paralle GC 策略。

（2）内存泄露。

程序中存在许多对象占用内存不能被回收，特别是大对象，导致频繁 Full GC 垃圾回收，而每次垃圾回收后又不能清理这些对象而回收占用空间，则系统的响应时间越长，当新对象多次申请空间时又不能满足需求，最终出现内存溢出而 WebLogic 挂起。

占用 CPU 高的进程主要是 Java 进程，即 WebLogic Server 运行进程，通过分析 JDK GC 日志，可以发现在 GC 垃圾回收占用系统资源严重，而 Full GC 垃圾回收又是整个垃圾回收的重点，而每次 Full GC 垃圾回收都是对那些在年轻代区域中不能被回收的对象进行回收。

同时结合观察，未进行 Full GC 时，系统的 CPU 使用正常；当每次在 FULL GC 期间，系统 CPU 都在高位，说明 CPU 高与 Full GC 垃圾回收有关。

3. 垃圾回收导致服务器挂起解决方法

（1）合理的 JVM 配置。

尽量采用静态的内存使用策略。

- -Xms=-Xmx。
- 设定-Xmn。
- -XX:MaxPermSize = -XX:PermSize。
- 合理设定-XX:SurvivorRatio。
- 慎用 Parallel GC 策略。
- 慎用-Xoptgc 参数。
- 使用–XX:+DisableExplicitGC，禁止 system.gc()的显式调用。
- TenuringThreshold 值的最大化。
- 使用最新版本的 JDK。

（2）修改程序。

内存溢出问题其彻底解决办法也只能修改程序，调整相关参数只能起到缓解的作用。

11.4.2　代码优化中服务器挂起

这个模式比较简单，顾名思义，当 JVM 在进行 Java 代码优化时，比如 JIT 编译时，会占用大量的 CPU 资源，有时对外表现为暂时没有响应。

一般情况下，优化做完了服务器即能自动恢复正常，所以这种情况顺其自然就可以，没有特别刻意地处理。

11.4.3　应用程序死锁导致服务器挂起

1．什么是应用程序死锁导致服务器挂起

由多线程带来的性能改善是以可靠性为代价的，主要是因为这样有可能产生线程死锁。线程死锁时，第一个线程等待第二个线程释放资源，而同时第二个线程又在等 待第一个线程释放资源。我们来想象这样一种情形：在人行道上两个人迎面相遇，为了给对方让道，两人同时向一侧迈出一步，双方无法通过，又同时向另一侧迈出一步，这样还是无法通过。双方都以同样的迈步方式堵住了对方的去路。假设这种情况一直持续下去，这样就不难理解为何会发生死锁现象了。

2．服务器挂起初始探查

毕竟应用程序死锁服务器挂起属于常规服务器挂起的一种，所以在服务器挂起时，有必要先进行“常规服务器挂起模式”中介绍的初始探查步骤，在确定是应用程序死锁问题后，再做一步的分析。

“常规服务器挂起模式”中介绍了初始探查步骤，这些步骤包括以下内容。

- 确定线程是被占用还是空闲。
- 以较短间隔进行若干 Thread dump。
- 确定线程正在执行的工作以及线程挂起的原因。

3．应用程序死锁服务器挂起探查

一般如果服务器挂起是由于应用程序死锁相关的问题引起时，在 Thread dump 里就会记录相关的信息，以下 Thread dump 分析就是一个死锁。

示例 11-4：

```
"[ACTIVE] ExecuteThread: '153' for queue: 'weblogic.kernel.Default
(self-tuning)'" daemon prio=1 tid=0x00002aab370ea700 nid=0x3eec waiting for
monitor entry [0x0000000045378000..0x000000004537bea0]
at weblogic.utils.classloaders.ChangeAwareClassLoader.loadClass
(ChangeAwareClassLoader.java:35)
- waiting to lock <0x00002aaabc69ddb0> (a weblogic.utils.classloaders.
ChangeAwareClassLoader)
```

大部分线程都在等待这个锁，而这个锁的所有者如下。

示例 11-5：

```
"[ACTIVE] ExecuteThread: '120' for queue: 'weblogic.kernel.Default
(self-tuning)'" daemon prio=1 tid=0x00002aab439c2f10 nid=0x2dee runnable
[0x00000000451f0000..0x00000000451f5e20]
    at java.lang.Throwable.fillInStackTrace(Native Method)
    - waiting to lock <0x00002aaab0873340> (a java.lang.ClassNot-
    FoundException)
    at java.lang.Throwable.<init>(Throwable.java:218)
    at java.lang.Exception.<init>(Exception.java:59)
    at java.lang.ClassNotFoundException.<init>(ClassNotFoundException.
    java:65)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    - locked <0x00002aaabadfc938> (a java.net.URLClassLoader)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
    - locked <0x00002aaabbf1cd20> (a weblogic.utils.classloaders.Generic-
    ClassLoader)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
    at weblogic.utils.classloaders.GenericClassLoader.loadClass(Generic-
    ClassLoader.java:161)
    at weblogic.utils.classloaders.FilteringClassLoader.findClass
    (FilteringClassLoader.java:83)
    at weblogic.utils.classloaders.FilteringClassLoader.loadClass
    (FilteringClassLoader.java:68)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
    - locked <0x00002aaabd5b2228> (a weblogic.utils.classloaders.Generic-
    ClassLoader)
```

```
at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
- locked <0x00002aaabc69ddb0> (a weblogic.utils.classloaders.
ChangeAwareClassLoader)
at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
at weblogic.utils.classloaders.GenericClassLoader.loadClass
(GenericClassLoader.java:161)
at weblogic.utils.classloaders.ChangeAwareClassLoader.loadClass
(ChangeAwareClassLoader.java:35)
- locked <0x00002aaabc69ddb0> (a weblogic.utils.classloaders.Change-
AwareClassLoader)
```

这个锁的所有者在等待另外一个锁，这个锁的所有者正是 ExecuteThread: 153，造成死锁。

4. 应用程序死锁服务器挂起解决方法

解决死锁没有简单的方法，这是因为使线程产生这种问题是很具体的情况，而且往往是在高负载的情况下才会出现。大多数软件测试产生不了足够多的负载，所以不可能暴露所有的线程错误。在每一种使用线程的语言中都存在线程死锁问题。由于使用 Java 进行线程编程比使用 C 容易，所以 Java 程序员中使用线程的人数更多，线程死锁也就更普遍。可以在 Java 代码中增加同步关键字的使用，这样可以减少死锁，但这样做也会影响性能。如果负载过重，数据库内部也有可能发生死锁。

如果程序使用了永久锁，比如锁文件，而且程序结束时没有解除锁状态，则其他进程可能无法使用这种类型的锁，既不能上锁，也不能解除锁。这会进一步导致系统不能正常工作，这时必须手动地解锁。

11.4.4 JDBC 中的服务器挂起

1. 什么是 JDBC 中的服务器挂起

在通过由应用程序或 WebLogic Server 本身使用的 JDBC 连接进行调用时，此连接会在整个调用期间内阻塞一个 WebLogic Server 执行线程。尽管在 SQL 查询上阻塞的线程需要等待，但 JVM 将通过其线程调度机制确保 CPU 获得可运行线程。但是，由 JDBC 调用占用的线程将保留给应用程序使用，直至该调用从 SQL 查询返回。

即使事务超时也不会终止由在此事务中登记的资源完成的任何操作，或者使其超时。这些操作将正常地运行，而不会出现中断。事务超时只是在事务上设置一个标记，将其标记为回滚，这样提交此事务的任何后续请求都将失败，系统抛出 TimedOutException 或 RollbackException。但是，如前所述，长时间运行 JDBC 调用会导致 WebLogic Server 执行线程阻塞，如果所有线程均被阻塞，没有能够处理传入请求执行线程，则最终可能导致实例挂起。

最新版本的 WebLogic Server 具有健全性检查功能，能够定期检查线程不响应的时间是否达到特定时长（默认值为 600 秒）。如果是这样，系统会向日志文件输出一条如下所

示的警告消息。

示例 11-6：

```
####<Nov 6, 2010 1:42:30 PM EST> <Warning> <WebLogicServer> <mydomain>
<myserver> <CoreHealthMonitor> <kernel identity> <>
<000337> <ExecuteThread: '64' for queue: 'default' has been busy for "740"
seconds working on the request "Scheduled Trigger",
which is more than the configured time (StuckThreadMaxTime) of "600"
seconds.>
```

这并不会中断线程，而只是提供给管理员的一项通知。阻塞线程恢复为正常状态的唯一途径是等待它正处理的请求完成。这种情况下，WebLogic Server 日志文件中将出现一条如下的消息。

示例 11-7：

```
####<Nov 7, 2010 4:17:34 PM EST> <Info> <WebLogicServer><mydomain>
<myserver> <ExecuteThread: '66' for queue: 'default'>
<kernel identity> <> <000339> <ExecuteThread: '66' for queue: 'default' has
become "unstuck".>
```

2. JDBC 中的服务器挂起探查

（1）以下是在 JDBC 调用时导致 WebLogic Server 实例挂起的各种可能原因。

- ❑ 在 JDBC 代码中使用 DriverManager.getConnection()。
- ❑ 发布给数据库的 SQL 查询返回时间异常。
- ❑ 配置了 JDBC 连接池的数据库挂起且不及时从调用返回。
- ❑ 低速或超载的网络导致数据库调用速度减慢或挂起。
- ❑ 死锁导致所有执行线程挂起和永久等待。
- ❑ JDBC 连接池中的 RefreshMinutes 或 TestFrequencySeconds 属性导致 WebLogic Server 中出现挂起期间。
- ❑ JDBC 连接池收缩和数据库连接的重新创建使响应时间变长。

（2）同步的 DriverManager.getConnection()。

旧版本的 JDBC 应用程序代码有时使用 DriverManager.getConnection() 调用来通过特定驱动程序取得数据库连接。不建议使用此项技术，因为它可能会导致死锁，或者至少相对降低连接请求的性能。究其原因，是因为所有 DriverManager 调用都采用类同步模式，也就是说一个线程中的一个 DriverManager 调用将阻塞一个 WebLogic Server 实例内任何其他线程中的所有其他 DriverManager 调用。

此外，SQL Exception 的构造器会构造一个 DriverManager 调用，而且大多数驱动程序使用 DriverManager.println()调用进行日志记录，这其中的任何一项都会阻塞发出 DriverManager 调用的所有其他线程。

DriverManager.getConnection()在返回为数据库建立的物理连接之前可能会需要相对较

长的时间。即使不发生死锁，所有其他调用也需要等到这个线程获得连接。在像 WebLogic Server 这样的多线程系统中，这不是最佳的方式。此外，我们的文档也明确指出不应使用 DriverManager.getConnection()。

如果愿意在 JDBC 代码中使用 JDBC 连接，应使用 WebLogic Server JDBC 连接池为其定义一个数据源，并从此数据源获得连接。这样您将享有池的所有优点（资源共享、连接重用、数据库关闭后的连接刷新等）。它还将帮助您避免 DriverManager 调用可能发生的死锁。

在 DriverManager.getConnection() 调用中阻塞的典型线程如下。

示例 11-8：

```
"ExecuteThread-39" daemon prio=5 tid=0x401660 nid=0x33 waiting for monitor
entry [0xd247f000..0xd247fc68]
  at java.sql.DriverManager.getConnection(DriverManager.java:188)
  at com.bla.updateDataInDatabase(MyClass.java:296)
  at javax.servlet.http.HttpServlet.service(HttpServlet.java:865)
  at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:120)
  at weblogic.servlet.internal.ServletContextImpl.invokeServlet
(ServletContextImpl.java:945)
  at weblogic.servlet.internal.ServletContextImpl.invokeServlet
(ServletContextImpl.java:909)
  at weblogic.servlet.internal.ServletContextManager.invokeServlet
(ServletContextManager.java:269)
  at weblogic.socket.MuxableSocketHTTP.invokeServlet
(MuxableSocketHTTP.java:392)
  at weblogic.socket.MuxableSocketHTTP.execute(MuxableSocketHTTP.
  java:274)
  at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:130)
```

（3）长时间运行的 SQL 查询。

长时间运行的 SQL 查询在其执行期间将阻塞执行线程，直至它们将结果返回给发出调用的应用程序。这就意味着，需要修改 WebLogic Server 实例的配置来处理应用程序负载要求的足够多的调用。这种情况的限制因素是执行线程数和 JDBC 连接池中的连接数。一般的经验方法是将池中的连接数设置为等于执行线程数，以便能够实现最优的资源利用。如果使用 JTS，则池中的可用连接应更多一些，因为某些连接可能会保留给实际处于非活动状态的事务。

对于在长时间运行的 SQL 调用期间挂起的线程，其在 Thread dump 中的堆栈与挂起的数据库的堆栈十分相似。

（4）挂起的数据库。

对于依赖于数据库的应用程序来说，良好的数据库性能是其性能的关键。因此，挂起的数据库可能会阻塞 WebLogic Server 实例中许多或所有可用的执行线程并最终导致服务器挂起。要诊断这一问题，应从挂起的 WebLogic Server 实例获得 5～10 个 Thread dump，

并检查您的执行线程（在默认队列或您的应用程序线程队列中）当前是否在 SQL 调用之中并在等待来自数据库的结果。当前发出 SQL 查询的线程的典型堆栈跟踪如下例所示。

示例 11-9：

```
"ExecuteThread: '4' for queue: 'weblogic.kernel.Default'" daemon prio=5
tid=0x8e93c8 nid=0x19 runnable [e137f000..e13819bc]
  at java.net.SocketInputStream.socketRead0(Native Method)
  at java.net.SocketInputStream.read(SocketInputStream.java:129)
  at oracle.net.ns.Packet.receive(Unknown Source)
  at oracle.net.ns.DataPacket.receive(Unknown Source)
  at oracle.net.ns.NetInputStream.getNextPacket(Unknown Source)
  at oracle.net.ns.NetInputStream.read(Unknown Source)
  at oracle.net.ns.NetInputStream.read(Unknown Source)
  at oracle.net.ns.NetInputStream.read(Unknown Source)
  at oracle.jdbc.ttc7.MAREngine.unmarshalUB1(MAREngine.java:931)
  at oracle.jdbc.ttc7.MAREngine.unmarshalSB1(MAREngine.java:893)
  at oracle.jdbc.ttc7.Oall7.receive(Oall7.java:375)
  at oracle.jdbc.ttc7.TTC7Protocol.doOall7(TTC7Protocol.java:1983)
  at oracle.jdbc.ttc7.TTC7Protocol.fetch(TTC7Protocol.java:1250)
  - locked <e8c68f00> (a oracle.jdbc.ttc7.TTC7Protocol)
  at oracle.jdbc.driver.OracleStatement.doExecuteQuery(OracleStatement.
  java:2529)
  at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout(Oracle-
  Statement.java:2857)
  at oracle.jdbc.driver.OraclePreparedStatement.executeUpdate
(OraclePreparedStatement.java:608)
  - locked <e5cc44d0> (a oracle.jdbc.driver.OraclePreparedStatement)
  - locked <e8c544c8> (a oracle.jdbc.driver.OracleConnection)
  at oracle.jdbc.driver.OraclePreparedStatement.executeQuery
(OraclePreparedStatement.java:536)
  - locked <e5cc44d0> (a oracle.jdbc.driver.OraclePreparedStatement)
  - locked <e8c544c8> (a oracle.jdbc.driver.OracleConnection)
  at weblogic.jdbc.wrapper.PreparedStatement.executeQuery(Prepared-
  Statement.java:80)
  at myPackage.query.getAnalysis(MyClass.java:94)
  at jsp_servlet._jsp._jspService(__jspService.java:242)
  at weblogic.servlet.jsp.JspBase.service(JspBase.java:33)
  at weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction.run
(ServletStubImpl.java:971)
  at weblogic.servlet.internal.ServletStubImpl.invokeServlet
   (ServletStubImpl.java:402)
  at weblogic.servlet.internal.ServletStubImpl.invokeServlet
   (ServletStubImpl.java:305)
  at weblogic.servlet.internal.RequestDispatcherImpl.includ
e(RequestDispatcherImpl.java:607)
```

```
    at weblogic.servlet.internal.RequestDispatcherImpl.include
(RequestDispatcherImpl.java:400)
    at weblogic.servlet.jsp.PageContextImpl.include(PageContextImpl.
    java:154)
    at jsp servlet. jsp.  landingbj. jspService(  landingbj.java:563)
    at weblogic.servlet.jsp.JspBase.service(JspBase.java:33)
    at weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction.run
(ServletStubImpl.java:971)
    at weblogic.servlet.internal.ServletStubImpl.invokeServlet(Servlet-
    StubImpl.java:402)
    at weblogic.servlet.internal.ServletStubImpl.invokeServlet(Servlet-
    StubImpl.java:305)
    at weblogic.servlet.internal.WebAppServletContext$ServletInvocation-
    Action.run
(WebAppServletContext.java:6350)
    at weblogic.security.acl.internal.AuthenticatedSubject.doAs
(AuthenticatedSubject.java:317)
    at weblogic.security.service.SecurityManager.runAs(SecurityManager.
    java:118)
    at weblogic.servlet.internal.WebAppServletContext.invokeServlet
(WebAppServletContext.java:3635)
    at weblogic.servlet.internal.ServletRequestImpl.execute(Servlet-
    RequestImpl.java:2585)
    at weblogic.kernel.ExecuteThread.execute(ExecuteThread.java:197)
    at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:170)
```

线程将处于运行状态。应比较不同 Thread dump 中的线程，查看它们是否及时接收 SQL 调用的返回结果或者它们是否在此同一调用中长时间挂起。如果 Thread dump 似乎指示 SQL 调用的响应时间较长，则应检查相应的数据库日志，查看是不是数据库中的问题导致这种执行速度缓慢或挂起的状况。

（5）低速网络。

WebLogic Server 与数据库之间的通信依赖于性能良好且可靠的网络来及时地处理请求。因此，网络性能低下可导致正在等待 SQL 查询结果的执行线程被挂起或阻塞。相关的堆栈跟踪将与上面挂起的数据库小节中的示例相似。仅仅通过分析 WebLogic Server Thread dump 不可能找到挂起或 SQL 查询速度低下的根本原因。它们给出了 SQL 调用的性能存在问题的第一个提示。下一步是检查是否存在导致 SQL 调用性能不佳的数据库网络或网络问题。

（6）死锁。

应用程序级的死锁与数据库级的死锁都可能导致线程挂起。您应检查 Thread dump，查看是否存在应用程序级的死锁。有关如何执行这一操作的信息在服务器挂—应用程序死锁模式中提供。数据库死锁可以在数据库日志中检测，或者可在 WebLogic Server 日志文件中找到的 SQL 异常。下面是相关 SQL 异常的一个示例。

示例 11-10：

```
java.sql.SQLException: ORA-00060: deadlock detected while waiting for
resource
  at oracle.jdbc.dbaccess.DBError.throwSqlException(DBError.java:170)
  at oracle.jdbc.oci8.OCIDBAccess.check_error(OCIDBAccess.java:1614)
  at oracle.jdbc.oci8.OCIDBAccess.executeFetch(OCIDBAccess.java:1225)
  at oracle.jdbc.oci8.OCIDBAccess.parseExecuteFetch(OCIDBAccess.
  java:1338)
  at oracle.jdbc.driver.OracleStatement.executeNonQuery(OracleStatement.
  java:1722)
  at oracle.jdbc.driver.OracleStatement.doExecuteOther(OracleStatement.
  java:1647)
  at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout
(OracleStatement.java:2167)
  at oracle.jdbc.driver.OraclePreparedStatement.executeUpdate
(OraclePreparedStatement.java:404)
```

数据库检测死锁并通过回滚引发死锁的一个或多个事务来解决死锁这一过程通常需要一些时间，因此在回滚结束之前，会有一个或多个执行线程被阻塞。

（7） RefreshMinutes 或 TestFrequencySeconds。

如果发现数据库性能低下、SQL 调用速度缓慢或连接高峰的时期反复出现，其原因可能在于 JDBC 连接池中 RefreshMinutes 或 TestFrequencySeconds 配置属性的设置。有关内容在探查 JDBC 故障模式中进行了详细介绍。除非 WebLogic Server 实例与数据库之间设有防火墙，否则您应禁用此功能。

（8）池收缩。

数据库的物理连接是应当打开一次并尽可能长时间保持打开的资源，因为新的连接请求对于数据库、操作系统内核及 WebLogic Server 而言是一个相当大的资源开销。因此，应在生产系统中禁用池收缩，使这一开销保持最小程度。如果启用池收缩，一旦对该池发出的连接请求不能得到满足，空闲的池连接就将被关闭，然后重新打开。

这些活动可能会需要一些时间，因此相关应用程序请求需要的时间可能会异常得长，使用户以为系统挂起。有关如何优化 JDBC 连接池配置的信息在探查 JDBC 故障模式中已提供。

3. JDBC 中服务器挂起的解决方案

有关如何分析挂起的 WebLogic Server 实例的一般信息在常规服务器挂起模式中提供。

大多数情况下，首先从挂起的系统获得 Thread dump 对于了解进展情况（例如不同的线程在做些什么以及它们为什么挂起）是非常有益的。通常，可以在生产系统上获得 Thread dump，但是对于很早以前的 JVM 版本（<1.3.1_09）则应小心，因为它们可能会在 Thread dump 期间崩溃。此外，如果 WebLogic Server 实例有大量线程，则意味着完成 Thread dump

需要一段时间，而其余线程将被阻塞。

进行多个 Thread dump（5～10 个），这些 Thread dump 彼此之间有若干秒钟的延迟，这使得您可以检查不同进程的进度情况。而且，它还将指示系统是否确实挂起（根本没有进度）或者吞吐速度是否极低，看起来像是系统已挂起。

分析 Thread dump 可以指示出实例的挂起是否确实是由于前一小节中提到的某一种原因。例如，如果所有线程都在一个 DriverManager 方法（如 getConnection()）中，则您已经确定出挂起的根本原因，并需要更改应用程序，以使用数据源或 Driver.connect()来代替 DriverManager.getConnection()。

11.4.5 EJB RMI 服务器挂起

1. 什么是 EJB RMI 服务器挂起

EJB RMI 服务器挂起是指有 EJB 或 RMI 问题造成的服务器挂起。实际上是由不合理的集群对象访问设计所致；由于设计不合理，虽然往往有本地对象可用，却过多使用了远程对象。其自身表现为许多线程都在长时间或无限期等待 RMI 请求或 JNDI 查找。

备注：由于 EJB RMI 服务器挂起是“常规服务器挂起”的问题的一部分，在进行排查之前，应先执行常规模式中的初始诊断步骤。

2. 服务器挂起初始探查

毕竟 EJB RMI 服务器挂起属于常规服务器挂起的一种，所以在服务器挂起时，有必要先进行“常规服务器挂起模式”中介绍的初始探查步骤，在确定是 EJB RMI 问题后，再做进一步的分析。

“常规服务器挂起模式”中介绍了初始探查步骤，这些步骤包括：确定线程是被占用还是空闲；以较短间隔进行若干 Thread dump；确定线程正在执行的工作以及线程挂起的原因。

3. EJB RMI 服务器挂起探查

一般如果服务器挂起是由于 EJB RMI 相关的问题引起时，在 Thread dump 里就会记录相关的信息，下面两个示例分别是 EJB RMI 服务器挂起时的 Thread dump 片段及完整信息。如果多个默认线程都有相同的调用栈模式，那么服务器挂起就有可能是因为远程 JNDI 查找某个对象引起的。

Thread dump 片断如下。

示例 11-11：

```
at weblogic.rjvm.ResponseImpl.waitForData(ResponseImpl.java:72)
. . .
weblogic.jndi.internal.WLContextImpl.lookup(WLContextImpl.java:341
. . .
at weblogic.rmi.internal.BasicOutboundRequest.sendReceive
 (BasicOutboundRequest.java:80
```

EJB RMI 挂起的完整 Thread dump 如下。

示例 11-12：

```
"ExecuteThread: '52' for queue: 'default'" daemon prio=5 tid=0x4b3e40b0
nid=0x1170 waiting on monitor [0x4c74f000..0x4c74fdbc]
 at java.lang.Object.wait(Native Method)
 at weblogic.rjvm.ResponseImpl.waitForData(ResponseImpl.java:72)
 at weblogic.rjvm.ResponseImpl.getTxContext(ResponseImpl.java:97)
 at weblogic.rmi.internal.BasicOutboundRequest.sendReceive
 (BasicOutboundRequest.java:80)
 at weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke(ReplicaAware-
 RemoteRef.java:262)
 at weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke(ReplicaAware-
 RemoteRef.java:229)
 at weblogic.rmi.internal.ProxyStub.invoke(ProxyStub.java:35)
 at $Proxy6.lookup(Unknown Source)
 at weblogic.jndi.internal.WLContextImpl.lookup(WLContextImpl.java:341)
 at weblogic.servlet.internal.HttpServer.lookupROIDS(HttpServer.java:789)
 at weblogic.servlet.security.internal.SecurityModule.getCurrentUser
 (SecurityModule. java:207)
 at weblogic.servlet.security.internal.SecurityModule.checkAuthenticate
 (SecurityModule. java:235)
 at weblogic.servlet.security.ServletAuthentication.weak(Servlet-
 Authentication.java:271
at weblogic.servlet.security.internal.ServletSecurityManager.checkAccess
(ServletSecurityManager.java:124)
at weblogic.servlet.internal.WebAppServletContext.invokeServlet(WebApp-
ServletContext. java:2518)
 at weblogic.servlet.internal.ServletRequestImpl.execute(ServletRequest-
 Impl.java:2260)
 at weblogic.kernel.ExecuteThread.execute(ExecuteThread.java:139)
at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:120)
```

发生 EJB RMI 服务器挂起时，线程像是已挂起，实际上是在等待远程 JVM 的响应；JNDI 查找发起请求；JNDI 查找会被发送到远程 JVM，因为发送的是全局查找请求。

在这个线程的栈追踪信息中，您马上就能看到一些相关的信息。线程处在某个等待方法中，也就是说等待发生某些事情，比如等待工作或数据的到达。这个线程调用了等待方法，因为它正等待来自 weblogic.rjvm.ResponseImpl.waitForData()方法中某个远程 JVM 的数据。如果栈中多个线程都在等待来自其他集群成员的数据的话，那么其他的集群成员很可能在等待来自这个 JVM 的数据。

继续分析调用栈，找出在 waitForData 之前正在进行什么操作，通常会发现正在进行 JNDI 查找。针对 JNDI 对象的查找将会被发送到远程 JVM。线程栈可能会显示应该从全局 JNDI 树上查找。从哪里获取 InitialContext 取决于应用程序代码衍生 InitialContext 的方式。

要想在全局 JNDI 树上查找某个对象，需要知道获取 InitialContext 的环境属性。作为

这些属性之一的提供程序，URL 会决定在哪个集群成员上获取 InitialContext 以及做后续的查找。如果没有提供任何属性的话，那么对 getInitialContext()的调用会使用 WLS 实例的当前环境，也就是使用本地的。在上面的线程栈中，设置了一个非集群成员实例的 provider URL。因此，线程栈显示出，将要做一个通过 rjvm sendReceive()方法的远程 JVM 调用。

总之，从 Thread dump 可以看出多个默认队列线程正在等待对某个 RMI 请求的响应。在每个 RMI 请求之前，发出了针对 weblogic.jndi.internal.WLContextImpl.lookup()的调用，从这可以看出应用程序代码正在尝试做 JNDI 查找。栈追踪信息中的下一个调用集显示出查找是一个远程的查找。



BasicOutboundRequest.sendReceive()和后续的 waitForData()。

4．EJB RMI 服务器挂起成因及解决方法

（1）EJB RMI 服务器挂起问题成因。

EJB RMI 服务器挂起问题如下。

① 基本上都出现在大量（EJB）线程等待远程响应时，即对远程 JVM 上的 JNDI 查找时。

② JNDI 查找使用与服务器建立的初始上下文，该上下文可能建立在本地服务器或远程服务器上。

③ 初始上下文。

可以与某个服务器建立初始上下文。

也可以与集群建立初始上下文，初始上下文会选择集群中的一个服务器，并允许访问部署到集群中的所有对象；但如果目标对象只绑定到一个服务器，则必须显示连接到主机服务器进行查找。

由此可以得出，问题出在初始上下文的创建位置（即是本地还是在远程地点创建）及创建原因！

（2）EJB RMI 服务器挂起解决方法步骤。要缓解问题，就需要减少远程查找的数量。

① 应用程序可能需要下列内容。

a．更充分利用本地上下文进行查找，即分析 PROVIDER_URL 的设置，该参数定义上下的建立位置。

b．要减少查找次数，也就是说，大量查找是否因为 JNDI 树中的对象或其他信息过多所导致的。

② 分析对象的部署方式和部署位置。

a．均匀部署。也就是部署到集群中的所有服务器，在该部署方式下，可以使用本地上下文查找，也可以使用集群上下文查找。

b．不均匀部署。也就是指部署到集群中的某些服务器，在该部署方式下，需要采取更有效的查找策略。

③ 解决方法综述。

在集群中，所有均匀部署的服务都在集群的全局 JNDI 中。所有集群成员都有相同的对象参照。相同的对象参照信息可以从本地 JNDI 树和全局树上获取。

重新检查 getInitialContext() 调用的代码，如果设置环境属性的话，特别是 PROVIDER_URL，就会强制进行远程查找。改写代码，以使用本地 JNDI 查找。

如果对象是不均匀部署的话，也就是只部署到了集群中的某些服务器上，可能就要求更好的查找策略，比如，均匀部署对象并使用本地或集群上下文及查找；部署额外的实例到更多的服务器上，并分散上下文/ 查找负载。

11.4.6　JSP 编译导致服务器挂起

1. 成因

主要是由于编译 JSP 过于频繁所导致的。

2. 解决方案

建议在应用中的 weblogic.xml 文件中添加如下参数。

示例 11-13：

```
<weblogic-web-app>
    <jsp-descriptor>
    <jsp-param>
    <param-name>keepgenerated</param-name>
    <param-value>true</param-value>
    </jsp-param>
    <jsp-param>
    <param-name>pageCheckSeconds</param-name>
    <param-value>-1</param-value>
    </jsp-param>
    </jsp-descriptor>
    <container-descriptor>
    <servlet-reload-check-secs>-1</servlet-reload-check-secs>
    </container-descriptor>
    <context-root>webpay</context-root>
</weblogic-web-app>
```

注 意

其中 pageCheckSecond 的默认值是 1，表示每隔 1 秒对 JSP 页面进行检查，检查 JSP 页面是否被修改、是否需要重新编译。系统上线后，需要对该默认值修改为–1，表示永不检查。修改为–1 的一个缺点是对于页面的修改需要重新部署整个 Web 应用。

第 12 章　异常高 CPU 占用率故障

12.1　异常高 CPU 占用率概述

12.1.1　回顾：进程、线程和 CPU 占用率

进程：以操作系统进程 ID（process ID, PID）来标识可以是单线程或多线程。

线程：是进程的子任务，与其他线程各占用一部分资源（如 CPU），可以映射到操作系统 PID，可以使用轻量型进程 ID（Light-Weight Process Id，LWPID）来识别，该 ID 与父进程关联。

CPU 占用率：计算机的 CPU 进程代表计算机上的多个进程、一个进程内的多个线程。

12.1.2　异常高 CPU 占用率的故障症状

异常高 CPU 占用率：当一个进程或线程占用的 CPU 资源百分百异常高时发生，可能会是其他进程或线程丧失或缺乏执行要求的处理任务所需的 CPU 处理能力，应进行定期检查（监视）。

异常高 CPU 占用率的故障症状：用户响应时间长，WebLogic 服务器运行速度异常缓慢，请求或操作开始出现超时。

12.2　异常高 CPU 占用率探查

12.2.1　探查概述

确定哪个（些）WLS 进程线程导致了异常高 CPU 占用率。发生高占用率情况时捕捉 Thread dump。找到服务器 Thread dump 中的高占用率线程（这一过程视所在平台而有所不同，该过程可能包括若干个步骤），重复执行上述过程来确保已建立了模式并找对了线程。利用一次或多次 Thread dump 探查发生异常高 CPU 占用率的具体原因。

12.2.2　在 Solaris 平台上探查

发生异常高 CPU 占用率时，可重复这些步骤来捕捉服务器活动的快照。

（1）使用以下命令捕捉哪些线程（LWPID）正在使用 CPU：prstat -L -p <WLSpid> 11。

（2）使用以下命令获得 LWPID 到（十进制）PID 的映射：pstack <WLSpid>。

（3）使用以下命令获得服务器 Thread dump：kill -3 <WLSpid>。

然后按下述方法利用收集到的输出。

（1）在 prstat 输出中找到使用率最高（最高频率条目）的 LWPID。

（2）在 pstack 输出中找到该 LWPID，获得对应的线程编号。

（3）将线程编号转换为十六进制号码。

（4）在服务器 Thread dump 中找到像 nid=<hexNum>这样的十六进制线程编号。

（5）确定该线程执行的哪一项任务导致了异常高 CPU 占用率。

Solaris prstat 输出示例如下。

示例 12-1：

```
$ prstat -L -p 9499 1 1
PID USERNAME SIZE RSS STATE PRI NICE TIME CPU PROCESS/LWPID
9499 usera 153M 100M sleep 58 0 0:00.22 0.6% java/8
9499 usera 153M 100M sleep 58 0 0:00.10 0.2% java/10
9499 usera 153M 100M sleep 58 0 0:00.11 0.1% java/9
9499 usera 153M 100M sleep 58 0 0:00.03 0.0% java/5
9499 usera 153M 100M sleep 58 0 0:01.01 0.0% java/1
9499 usera 153M 100M sleep 58 0 0:00.00 0.0% java/12
9499 usera 153M 100M sleep 58 0 0:00.00 0.0% java/11
9499 usera 153M 100M sleep 58 0 0:00.00 0.0% java/14
9499 usera 153M 100M sleep 58 0 0:00.00 0.0% java/13
9499 usera 153M 100M sleep 59 0 0:00.07 0.0% java/7
9499 usera 153M 100M sleep 59 0 0:00.00 0.0% java/6
9499 usera 153M 100M sleep 59 0 0:00.00 0.0% java/4
9499 usera 153M 100M sleep 58 0 0:00.11 0.0% java/3
9499 usera 153M 100M sleep 58 0 0:00.00 0.0% java/2
```

说明：在上面的示例中可以看出 LWPID 8 cpu 占用率最高。

Solaris pstack 输出示例如下。

示例 12-2：

```
----------------- lwp# 8 / thread# 76 --------------------
ff29d190 poll (e2e81548, 0, bb8)
ff24d154 select (0, 0, 0, e2e81548, ff2bf1b4, e2e81548) + 348
ff36b134 select (0, bb8, 7fffffff, fe4c8000, 0, bb8) + 34
fe0f62e4 __1cCosFsleep6FpnGThread_xl_i_ (0, bb8, fe4c8000, 1, 0, 1e2fd8)
+ 234
fe23f050 JVM_Sleep (2, 0, bb8, fe4de978, fe4c8000, 1e2fd8) + 22c
0008f7ac ???????? (e2e818d4, bb8, 1e2fd8, 984a4, 0, 109a0)
0008c914 ???????? (e2e8194c, 1, fe4d6a80, 98564, 8, e2e81868)
fe5324e8 __1cMStubRoutinesG_code1_ (e2e819d8, e2e81c10, a, f6cb5000, 4,
e2e818f0) + 3ec
fe0cbe94 __1cJJavaCallsLcall_helper6FpnJJavaValue_pnMmethodHandle_
```

```
pnRJavaCallArguments_pnGThread__v_ (e2e81c08,fe4c8000, e2e81b54, 1e2fd8,
8e764, e2e81c10) +308
fe1f6dbc __1cJJavaCallsMcall_virtual6FpnJJavaValue_nLKlassHandle_
nMsymbolHandlee81c08, e2e81b54) + 150pnGThread__v_(f6cb64b8, e2e81b40,
e2e81b44, fe4c8000, e2d8) + 60e 5pnGThread v (e2e81c08, e2e81c04,
e2e81c00,e2e81bf4, e2e81bec, 1e2f8000, e2e81d10, 1e, e) + 120FpnKJavaThread_
pnGThread__v_ (f6817ff8, 1e2fd8, fe4c 7fd70) + 3d8cKJavaThreadDrun6M_v_
(e2e02000, fe4d3e34, fe4c8000, 7fd70,1e2fd8,
fe213ec8 _start (fe4c8000, fe625d10, 0, 5, 1, fe401000) + 20
ff36b728 _thread_start (1e2fd8, 0, 0, 0, 0, 0) + 40
```

说明：运行 pstack 命令得到一个关于 LWPID 和 PID 的映射。例如，pstack 9499 并重定向到一个输出文件中。在上面的示例中 pstack 命令输出 lwp#8 映射到 thread#76。

Solaris Thread dump 示例如下。

示例 12-3：

```
$ kill -3 p 9499
. . .
"Thread-6" prio=5 tid=0x1e2fd8 nid=0x4c waiting on monitor [0xe2e81000..
0xe2e819d8]
at java.lang.Thread.sleep(Native Method)
at weblogic.management.deploy.GenericAppPoller.run(GenericAppPoller.
java:139
```

说明：由于 lwp#8 映射到 thread#76，您需要将十进制的 76 转换成十六进制格式，即 0x4c。执行 kill-3 <WLSpid>得到 Thread dump,在 Thread dump 中找到本地线程 nid=0x4c。在这个示例里，占用 CPU 最高的线程正在 sleeping，这个 Thread dump 并没有及时补货线程活动状态即线程正在执行的操作。从 Thread dump 中可以判断出现场执行什么操作会引起高 CPU 利用。

为快速、重复的捕捉数据，可编写一个脚本，下边是脚本示例。

示例 12-4：

```
for loopnum in 1 2 3
do
prstat -L -p $1 1 1 >> dump_high_cpu.txt
pstack $1 >> dump high cpu.txt
kill -3 $1
echo "prstat, pstack, and thread dump done. #" $loopnum
sleep 1
echo "Done sleeping."
done
```

说明：携带参数（WLS 进程的 PID），重复执行 3 次。此脚本可将 prstat 和 pstack 信息追加到文件 dump_high_cpu.txt 中。Thread dump 信息会出现在将 stdout 重定向到的文件中或输出到屏幕上。

12.2.3　在 HP-UX 平台上探查

下载 BEA 技术支持部门开发的 hp_prstat 使用程序。可参阅 support.bea.com High CPU Usage Parren。

发生异常高 CPU 占用率时，可重复这些步骤来确定服务器活动的模式。

（1）使用以下命令捕捉那些正在使用 CPU 的线程 (LWPID)：hp_prstat <WLSpid>。

（2）使用以下命令获得服务器 Thread dump：kill -3 <WLSpid>。

然后按下述方法利用收集到的输出。

（1）找到“用户时间”增加最快的 1 个或多个 LWPID。

（2）在服务器 Thread dump 中找到像 lwp_id=<LWPID>这样的对应 LWPID 号码。

（3）确定该线程执行的哪一项任务导致了异常高 CPU 占用率。

HP-UX hp_pstat 示例如下。

示例 12-5：

```
$ hp_prstat  4426
lwpid pid pri status UsrTime SysTime
285365 4426 154 1 29 3
. . .
285415 4426 154 1 0 7
285416 4426 154 1 0 7
285417 4426 154 1 0 7


. . .
$ hp prstat  4426
lwpid pid pri status UsrTime SysTime
285365 4426 154 1 29 3
. . .
285415 4426 154 1 0 7
285416 4426 154 1 13 7
285417 4426 154 1 0 7
```

说明：查找 UsrTime 增长最快的那一行获取 LWPID。（如果有多个增长速度较快的可疑线程，则全部取出。）

HP-UX Thread dump 示例如下。

示例 12-6：

```
$ kill -3 4426
"Thread-6" prio=8 tid=0x0004f620 nid=75 lwp_id=285475 waiting on monitor
[0x66d5e000..0x66d5e500]
at java.lang.Thread.sleep(Native Method)
at weblogic.management.deploy.GenericAppPoller.run
(GenericAppPoller.java:139)
```

```
"ExecuteThread: '11' for queue: 'default'" daemon prio=10 tid=0x0004ad00
nid=23 lwp_id=285416 runnable [0x67874000..0x67874500]
at java.net.SocketOutputStream.socketWrite(Native Method)
at java.net.SocketOutputStream.write(Unknown Source)
. . .
at examples.servlets.HelloWorldServlet.service
 (HelloWorldServlet.java:28)
at javax.servlet.http.HttpServlet.service
(HttpServlet.java:853)
```

说明：从线程快照中找到之前取得的 LWPID，检查是什么原因引起 CPU 的高利用率，如果线程是 WebLogic 自身的线程，可联系 BEA 客服支持。

12.2.4 在 Linux 平台上探查

在 Linux 系统上，每个线程均作为独立的进程出现。

发生异常高 CPU 占用率时，可重复这些步骤来捕捉服务器活动的快照。

（1）使用 top 确定哪些线程正在使用 CPU。

（2）使用以下命令获得服务器 Thread dump: kill-3 <WLSpid>。

然后按下述方法利用收集到的输出。

（1）在 top 输出中，寻找具有启动服务器的 userID 的进程线程。

（2）获得 CPU 占用率最高的服务器线程的 PID。

（3）将高占用率的 PID 转换为十六进制值。

（4）在服务器 Thread dump 中找到十六进制 PID。

（5）确定该线程执行的哪一项任务导致了异常高 CPU 占用率。

Top 输出示例如下。

示例 12-7：

```
PID USER PRI NI SIZE RSS SHARE STAT %CPU %MEM TIME COMMAND
...
22962 adminWLS 9 0 86616 84M 26780 S 0.0 4.2 0:00 java
...
```

说明：在 Linux 上每个线程被映射成为一个进程，这点和其他类 UNIX 系统不同。上面给出的只是 top 命令输出的很小的一段内容。

Thread dump 输出示例如下。

示例 12-8：

```
. . .
"ExecuteThread: '0' for queue: 'default'" daemon prio=1 tid=0x83da550
nid=0x59b2 waiting on monitor [0x56138000..0x56138870]
at java.lang.Object.wait(Native Method)
at java.lang.Object.wait(Object.java:415)
```

```
    at weblogic.kernel.ExecuteThread.waitForRequest(ExecuteThread.java:146)
    at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:172)
    . . .
```

说明：如果 PID 是 22962 时转换成十六进制为 0X59B2。查询 Thread dump 找到 nid=0X59B2 的线程。例如上面的示例 0X59B2 对应执行线程 0。在上面的例子中 CPU 利用率为 0%。

为了快速、重复地捕捉数据，可编写一个脚本。如下所示是脚本示例。

示例 12-9：

```
    for loopnum in 1 2 3
    do
    top -b -n1 >> dump high cpu.txt
    kill -3 $1
    echo "cpu snapshot and thread dump done. #" $loopnum
sleep 1
    echo "Done sleeping."
    done
```

说明：携带参数（WLS 进程的 PID），重复执行 3 次。此脚本把 top 信息追加到名为 dump_high_cpu.txt 的文件中。Thread dump 信息会出现在将 stdout 重定向到的文件中或输出到屏幕上。

12.2.5　在 AIX 平台上探查

发生异常高 CPU 占用率时，可重复这些步骤来捕捉服务器活动的快照。

（1）找到正在使用 CPU 的线程 ID(Thread ID, TID)：ps -mp <WLSpid> -o THREAD 即 CP 值最高的线程。

（2）使用以下命令获得服务器 Thread Dump：kill -3 <WLSpid>。

然后按下述步骤进行探查。

（1）使用以下命令在服务器进程上运行 dbx：dbx -a <WLSpid>。

（2）在 dbx 中，使用以下命令获得所有线程的列表：thread。

（3）在输出线程列表中，找到高占用率 TID 及其号码（$t<num> 形式的号码）。

（4）在 dbx 中，使用以下命令获得详细的线程信息：th info <num>。

（5）在 general 输出中找到 pthread_t（十六进制）值。

（6）重要说明：从进程中分离 detach

（7）在服务器 Thread dump 中找到像 native ID:<hexNum>这样的十六进制 phtread_t 号码。

（8）确定该线程执行哪一项任务导致了异常高 CPU 占用率。

PS 命令输出示例如下。

示例 12-10：

```
$ ps -mp 250076 -o THREAD
USER PID PPID TID ST CP PRI SC WCHAN F TT BND COMMAND
usera 250076 217266 - A 38 60 72 * 242011 pts/0 - /landingbj /sharedInstalls/
aix/jdk130/...
- - - 315593 Z 0 97 1 - c00007 - - -
- - - 344305 S 0 60 1 f1000089c020e200 400400 - - -
- - - 499769 S 0 60 1 f1000089c0213a00 400400 - - -
. . .
- - - 655429 S 0 60 1 f10000879000a040 8410400 - - -
- - - 659527 S 0 60 1 f10000879000a140 8410400 - - -
- - - 663625 S 0 60 1 f10000879000a240 8410400 - - -
- - - 667723 S 37 78 1 f1000089c020f150 400400 - - -
- - - 671821 S 0 60 1 f10000879000a440 8410400 - - -
- - - 675919 S 0 60 1 - 418400 - - -
. . .
```

说明：执行命令 ps -mp <WLSpid> -o THREAD 查看相关内容。注意 TID667723 在 CP 列数值达到 37 而其他线程接近 0。

AIX dbx thread 示例如下。

示例 12-11：

```
$dbx -a 250076
(dbx) thread
thread state-k wchan state-u k-tid mode held scope function
. . .
$t15 wait 0xf10000879000a140 blocked 659527 k no sys _event_sleep
$t16 wait 0xf10000879000a240 blocked 663625 k no sys _event_sleep
$t17 run running 667723 k no sys JVM_Send
$t18 wait 0xf10000879000a440 blocked 671821 k no sys _event_sleep
$t19 wait running 675919 k no sys poll
$t20 wait 0xf10000879000a640 blocked 680017 k no sys _event_sleep
. . .
```

说明：执行命令 dbx -a 250076。在 dbx 中列出所有本地线程执行命令：thread。上面列出的只是有关线程的一小部分。

AIX dbx th info 示例如下。

示例 12-12：

```
(dbx) th info 17
thread state-k wchan state-u k-tid mode held scope function
$t17 run running 667723 k no sys JVM_Send
general:
   pthread addr = 0x3ea55c68 size = 0x244
   vp addr = 0x3e69e5e0 size = 0x2a8
   thread errno = 2
```

```
    start pc = 0x300408b0
    joinable = no
    pthread_t = 1011
  scheduler:
    kernel =
    user = 1 (other)
  event :
    event = 0x0  cancel = enabled, deferred, not pending
  stack storage:
  . . .
  (dbx)detach
```

说明：为了得到本地线程的信息，执行命令 th info 17。

AIX Thread Dump 示例如下。

示例 12-13：

```
"ExecuteThread: '11' for queue: 'default'" (TID:0x31cf86d8, sys_thread_t:
0x3e5ea108, state:R, native ID:0x1011) prio=5
at java.net.SocketOutputStream.socketWrite(Native Method)
. . .
at java.io.PrintWriter.println(PrintWriter.java(Compiled Code))
at examples.servlets.HelloWorldServlet.service(HelloWorld
Servlet.java(Compiled Code))
at javax.servlet.http.HttpServlet.service
(HttpServlet.java:853)
at weblogic.servlet.internal.ServletStubImpl$Servlet
InvocationAction.run(ServletStubImpl.java:1058)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet (ServletStubImpl.
java:401)
 at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:306)
. . .
```

说明：拿到 pthread_t 号并用它在 Thread dump 找到正确的线程。在 dbx 中执行 detach，从 WebLogic 进程中断开。拿到 pthread_t，用它在 Thread dump 中来找到准确的线程，并查看其状态。

AIX thread dump 示例如下。

示例 12-14：

```
"ExecuteThread: '11' for queue: 'default'" (TID:0x31cf86d8, sys_thread_t:
0x3e5ea108, state:R, native ID:0x1011) prio=5
at java.net.SocketOutputStream.socketWrite(Native Method)
. . .
at java.io.PrintWriter.println(PrintWriter.java(Compiled Code))
at examples.servlets.HelloWorldServlet.service(HelloWorld
```

```
Servlet.java(Compiled Code))
at javax.servlet.http.HttpServlet.service
(HttpServlet.java:853)
at weblogic.servlet.internal.ServletStubImpl$Servlet
InvocationAction.run(ServletStubImpl.java:1058)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:401)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:306)
. . .
```

说明：从上步中取得的 pthread_t 所匹配的 native ID 在 Thread dump 中找到相应线程。

12.2.6 在 Windows 平台上探查

下载一个工具来分析线程的 CPU 的占用占用率，pslist 可提供进程的详细信息。Process Explorer 是一个可显示进程的动态性能统计信息的图形化工具。

发生异常高 CPU 占用率时，可重复这些步骤来确定服务器的线程活动。

（1）使用以下两种工具之一捕捉正在使用 CPU 的线程（LWPID）：pslist -d <WLSpid> 或 Process Explorer。

注：下载地址:_http://www.sysinternals.com/ntw2k/freeware/pslist.shtml 和 http://www.sysinternals.com/ntw2k/freeware/procexp.shtml。

（2）使用以下命令获得服务器 Thread dump：Control+Break。

然后按下述方法利用收集到的输出。

（1）找到“用户时间”和“内核时间”增加最快的线程。

（2）将线程编号转换为十六进制值，如<hexNum>。

（3）在服务器 Thread dump 中找到像 nid=<hexNum>这样的十六进制线程编号。

（4）确定该线程执行的哪一项任务导致了异常高 CPU 占用率。

pslist 输出示例如下。

示例 12-15：

```
$ pslist -d 1720
java 1720:
Tid Pri Cswtch State User Time Kernel Time Elapsed Time
1520 8 9705 Wait:UserReq 0:00:23.734 0:00:01.772 0:08:14.511
1968 8 6527 Wait:UserReq 0:00:06.309 0:00:00.070 0:08:14.120
1748 15 157 Wait:UserReq 0:00:00.010 0:00:00.010 0:08:14.110
. . .
588 10 59123 Wait:UserReq 0:00:48.830 0:00:02.633 0:08:01.211
1784 8 150 Wait:UserReq 0:00:00.090 0:00:00.000 0:08:01.201
1756 8 251 Wait:UserReq 0:00:00.941 0:00:00.000 0:08:01.201
1716 8 6 Wait:Queue 0:00:00.000 0:00:00.000 0:08:01.191
1800 8 1457 Wait:Queue 0:00:00.761 0:00:00.210 0:08:01.191
```

```
. . .
```

说明：在 WebLogic 进程上执行命令：pslist -d 1720。一段时间之后执行相同操作，对比一段时间之内 user/Kernel time 增长最快的线程。在上面的例子中 ID 588 号线程出现问题。拿到线程号 588 转换成十六进制及 0X24C。在 Thread dump 中查找 nid=0x24c 号线程。

Windows 平台下 Thread dump 示例如下。

示例 12-16：

```
"ExecuteThread: '10' for queue: 'default'" daemon prio=5 tid=0x20d75808
nid=0x24c runnable [219ff000..219ffd90]
at java.net.SocketOutputStream.socketWrite0(Native Method)
. . .
at java.io.PrintWriter.println(PrintWriter.java:515)
- locked <0x11d0d1c0>
(a weblogic.servlet.internal.ChunkWriter)
at examples.servlets.HelloWorld2.service(HelloWorld2.
java:94)
at javax.servlet.http.HttpServlet.service
(HttpServlet.java:853)
at weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction.run
(ServletStubImpl.java:1058)
. . .
```

12.3　异常高 CPU 占用率故障排除策略及相关资源

（1）通过重复收集信息确定占用率模式。

① 捕捉 CPU 占用率高的线程或用户时间快速增加的线程的快照。

② 同时获得 Thread dump。

③ 使用各平台提供的工具。

（2）在服务器的一个或多个 Thread dump 中查找异常高 CPU 占用率线程。

（3）在 Thread dump 中探查导致异常高 CPU 占用率的具体原因。

（4）继续监视，看是否还会发生故障。

第 13 章　执行线程丢失故障

13.1　WLS 的执行线程

WLS 执行流程示意图如图 13-1 所示。

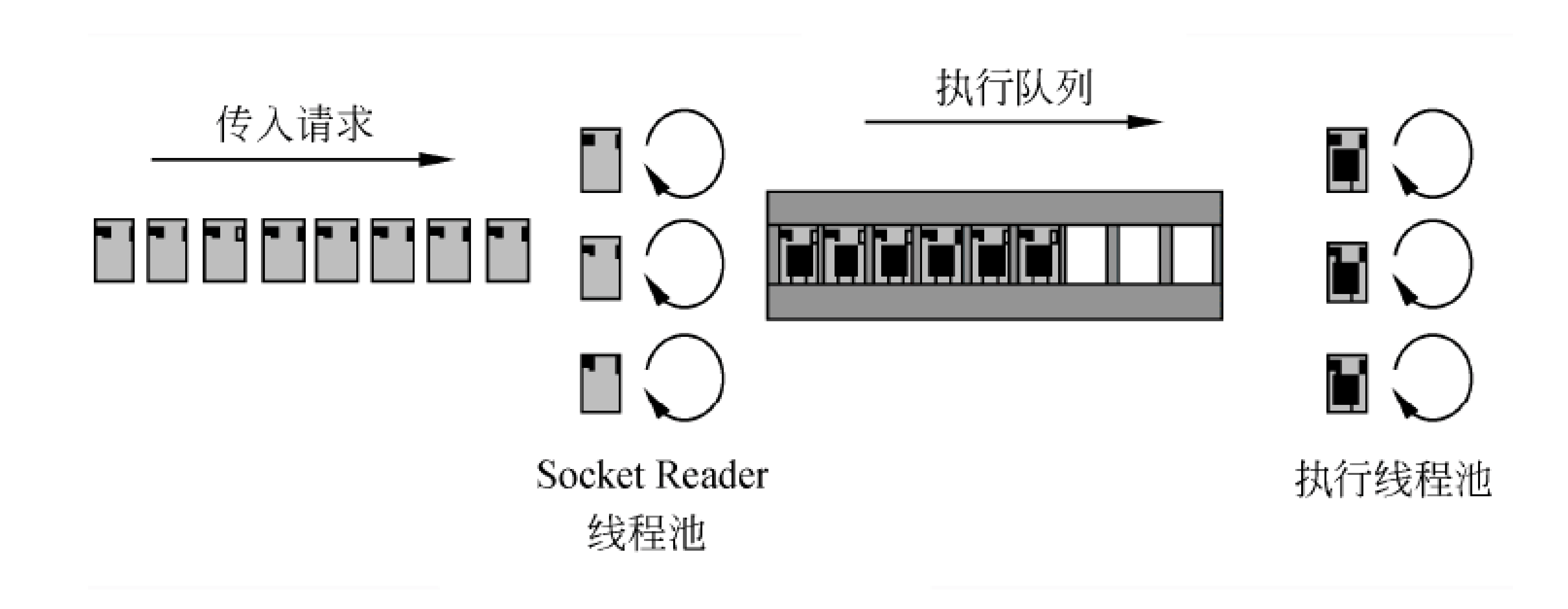


图 13-1

WLS 实例在线程池中执行所有处理，线程有如下分工。

执行线程：负责处理用户请求，执行应用程序要求的具体任务。默认个数为 15。

Socket Reader 线程：负责从 Socket 读取信息，处理网络通信量。默认个数为执行线程数的 33%。

Socket Reader 线程负责接收进来的请求，然后将其放入执行队列中，最后由执行线程执行具体的任务。

WLS 线程在某些情况下可能会出现丢失现象，正常情况下 WLS 的 Thread dump 信息中应该显示所有的线程运行信息。

应用程序也可以创建线程，用于执行特定的工作，用户创建的执行线程也可能发生丢失现象。

13.2　丢失线程时的故障症状

13.2.1　故障症状概述

线程丢失时，可能会出现下面的这些故障症状。

（1）可能抛出了未捕获的异常或错误。

（2）通常不会向服务器日志输出任何异常消息、堆栈跟踪或通知。

（3）会从服务器的 Thread dump 中消失。

（4）当某些线程等待来自其他（已消失的）线程的响应时，可能会导致服务器挂起。

（5）可能会导致服务器挂起，并因此被检测到。

（6）可能会导致意外超时或其他的异常行为。

（7）可能是应用程序创建的线程，这些线程会在许多方面对应用程序产生影响。

一般，线程丢失问题会在分析服务器挂起、未解释的超时或其他令人费解和未解释的行为时被发现。当线程抛出未捕获的异常或错误时，线程可能就会消失。这可能会使服务器挂起，因为可能有其他线程正在等待针对 monitor 的、但是永远不会被调用的 notify()，因为本应调用 notify 的线程消失了，导致其他线程一直在等待就有可能出现挂起现象。

13.2.2 线程丢失时 Thread dump 信息示例

显示线程 13 和线程 11 丢失的 Thread dump 示例如下。

示例 13-1：

```
"ExecuteThread: '14' for queue: 'default'" daemon prio=5 tid=0x7bc140
nid=0x13c runnable [0x157ef000..0x157efdc0] at weblogic.socket.NTSocket-
Muxer.getNextSocket(Native Method) at weblogic.socket.NTSocketMuxer.
processSockets(NTSocketMuxer.java:589) at weblogic.socket.SocketReaderRequest.
execute(SocketReaderRequest.java: 24) at weblogic.kernel.ExecuteThread.
execute(ExecuteThread.java:139) at weblogic.kernel.ExecuteThread.run
(ExecuteThread.java:120)
"ExecuteThread: '12' for queue: 'default'" daemon prio=5 tid=0x7bb8e0
nid=0xc5 r unnable [0x1576f000..0x1576fdc0] at weblogic.socket.NTSocketMuxer.
getNextSocket(Native Method) at weblogic.socket.NTSocketMuxer.processSockets
(NTSocketMuxer.java:589) at weblogic.socket.SocketReaderRequest.execute
(SocketReaderRequest.java: 24) at weblogic.kernel.ExecuteThread.execute
(ExecuteThread.java:139) at weblogic.kernel.ExecuteThread.run(ExecuteThread.
java:120)
"ExecuteThread: '10' for queue: 'default'" daemon prio=5 tid=0x7b90c0
nid=0x1a2 runnable [0x156ef000..0x156efdc0] at weblogic.socket.NTSocketMuxer.
getNextSocket(Native Method) at weblogic.socket.NTSocketMuxer.processSockets
(NTSocketMuxer.java:589) at . . .
```

以上信息是出现线程丢失的 Thread dump 示例，由于只出现了执行线程 10、12、14 的信息，而没有执行线程 11 和 13 的信息，所以可以判断执行线程 11 和 13 丢失（异常退出）了。

13.3 线程丢失原因及相应的解决方法分析

13.3.1 线程丢失原因概述

线程丢失的可能原因包括以下几方面。

（1）出现内存不足情况，这种情况会导致线程终止时得不到通知。

（2）处理异常时可能会出现问题。

（3）未捕获的异常或错误。

下面来针对不同的情况，一步步进行分析。

13.3.2　JVM 堆内存不足造成的线程丢失

如果通过分析，发现 JVM 堆内存不足是原因，那么具体分析如下。

（1）可能会抛出 OutOfMemoryException，而在处理该异常访问时又再次抛出同一异常，因而导致线程退出！

（2）此时可检查：JVM 堆的最大大小；实际使用的堆大小；堆内分配的永久生成空间的大小。

（3）使用以下参数将生成永久空间，并设置为 128MB 或更大：

```
-XX:MaxPermSize=128m
```

如果在设置了 MaxPermSize 以后，问题解决了，那么可能的解释是不足的 MaxPermSize 设置在某种程度上造成了 OutOfMemoryException 异常。

补充：设置 JVM 的内存大小。

① 在 commEnv 中设置。

找到在..\bea\weblogic92\common\bin 目录下的 commEnv.cmd 文件，用文本编辑器编辑，会看到 BEA 或是 Sun 的 JDK，选择自己所用的 JDK 修改一下代码段（以 Sun JDK 为例），重启生效。

示例 13-2：

```
:sun
if "%PRODUCTION_MODE%" == "true" goto sun_prod_mode
set JAVA_VM=-client
set MEM_ARGS=-Xms32m -Xmx200m -XX:MaxPermSize=128m -XX:+UseSpinning
set JAVA_OPTIONS=%JAVA_OPTIONS% -Xverify:none
goto continue
:sun_prod_mode
set JAVA_VM=-server
set MEM_ARGS=-Xms32m -Xmx200m -XX:MaxPermSize=128m -XX:+UseSpinning
goto continue
```

② 在 setDomainEnv.cmd 中设置。

找到在..\bea\user_projects\domains\landingbj(域名)\bin 目录下的 setDomainEnv.cmd 文件，修改方法同上。

示例 13-3：

```
if "%JAVA_VENDOR%"=="Sun" (
    set MEM_ARGS=%MEM_ARGS% %MEM_DEV_ARGS% -XX:MaxPermSize=128m
)
```

13.3.3　应用程序的异常处理造成的线程丢失

在排查问题时，还要分析应用程序的异常处理，因为应用程序的异常处理也可能会造成线程丢失现象出现。

（1）如果应用程序创建的线程丢失，则可略微缩小搜索范围。

（2）执行代码审核，以充分了解错误的处理方法，特别是对复合错误的处理方法。

（3）检查异常处理代码处理后续异常的方法。

（4）使用调试器检查代码，查找未捕获的异常。

另外，在必要时可修改代码，以免异常处理的方法不正确。

13.4　故障排除检查清单

故障排除策略如下。

（1）使用服务器 Thread dump 确定线程是否消失。

（2）探查任何 JVM 堆内存不足状态，并根据需要进行矫正。

（3）分析应用程序的异常处理，使用调试器和/或更正任何错误的处理方法。

（4）检查是否有与线程消失有关的任何其他异常事件。

（5）继续监视，看是否还会发生故障。

第 14 章　服务器 core dump 分析

14.1　什么是服务器 core dump 文件

（1）如果服务器的 Java 虚拟机（Java Virtual Machine，JVM）进程异常退出，操作系统就会产生二进制核心转储（core）文件。

（2）服务器 core dump 文件是进程在故障点的快照转储。

（3）在 UNIX 系统中，会在服务器的启动目录中创建服务器 core 文件。

（4）在 Windows 系统中，会在 c:\Documents and Settings\AllUsers\Documents\DrWatson 中产生类似文件。

14.2　什么情况下可以导致 core dump 文件的生成

- ❑ 服务器崩溃。
- ❑ JVM 崩溃。
- ❑ 执行本地（C 或 C++）代码失败时可导致服务器的 JVM 失败，从而产生服务器 core 文件。
- ❑ 计算机崩溃。
- ❑ HotSpot 错误。

补充：

（1）进程或文件资源限制可能会阻止产生 core 文件。

（2）core 文件是探查服务器故障原因所必需的。

（3）本地代码出现在以下位置。

① WebLogic Server 本地性能包。

② 2 类 JDBC 驱动程序。

③ 使用 Java 本地接口（Java Native Interface，JNI）调用来访问库的应用程序代码。

④ JVM 本身。

14.3　服务器 core dump 探查

14.3.1　探查概述

（1）确认服务器的 JVM 产生了 core 文件。

（2）使用 core 文件进行探查。

① 使用调试器获得堆栈跟踪。

② 确定导致故障的本地代码类型。例如，是 JDBC 驱动程序，还是应用程序代码的 JNI 调用。

（3）探查具体的故障成因。

① 这项工作可能要求探查者熟悉本地代码。

② 附加的调试记录可能会有帮助。

③ 排除法或替换法可能是确定真正成因的唯一手段。

（4）获得 Thread dump。

14.3.2　探查 Solaris 系统

（1）确认二 core 文件由 JVM 产生，也就是运行$ file <fullpath>/core。

（2）检查 Java 开发工具包（Java Development Kit, JDK）版本。

（3）使用调试器获得堆栈跟踪和线程信息：

① Sun 技术支持部门推荐使用 dbx。

② 使用 GNU 的 gdb 也可能会获得更有用的信息。

（4）确定导致故障的本地代码类型。

（5）使用 dbx。

确认 JDK 版本代码如下。

示例 14-1：

```
   $ java -version                     （获得 JDK 版）
   $ ls /opt/bin/dbx 或 which dbx      （获得 dbx 位置）
$ export DEBUG_PROG=/opt/bin/dbx       （设置 dbx 位置）
   启动 dbx：
$ <path to java command>/java corefile （对于 JDK1.3.X）
   或
$ dbx <path to java command>/java corefile（对于 JDK1.4.X）
   输入 dbx 命令：
   (dbx) where                         （显示堆栈摘要）
   (dbx) threads             （显示现有线程的状态）
   (dbx) quit                （退出 dbx）
```

14.3.3　探查 Linux 系统

（1）确认 core 文件由 JVM 产生，即运行$ file <fullpath>/core。

（2）检查 JDK 版本。

（3）使用 GNU 的最新版本 gdb 调试器获得堆栈跟踪和线程信息。

（4）确定导致故障的本地代码类型。

（5）使用 GNU gdb。

确认 JDK 版本代码如下。

示例 14-2：

```
$ java -version                         （获得 JDK 版本）
$ ls /usr/local/bin/gdb                 （获得 gdb 的位置）
$ export DEBUG_PROG=/usr/local/bin/gdb （设置为 gdb 的置）
启动 gdb
$ <path to java command>/java  corefile（对于 JDK1.3.X）
或
$ gdb <path to java command>/java corefile(对 JDK1.4.X)
输入 gbd 命令：
(gdb) where                 （显示堆栈摘要）
(gdb) thr                   （切换线程或显示当前线程）
(gdb) info thr              （查询现有线程信息）
(gdb) thread apply 1 bt     （向回跟踪到 thread #1）
(gdb) quit                  （退出 gdb）
```

（6）使用 gdb bt 输出实例。

示例 14-3：

```
[root@app10 dennis]# gdb java core.11751
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(no debugging symbols found)
Reading symbols from /lib64/libpthread.so.0...(no debugging symbols
found)...done.
Loaded symbols for /lib64/libpthread.so.0
Reading symbols from /usr/java/jre1.6.0 19/lib/amd64/jli/libjli.so...(no
debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0_19/bin/../lib/amd64/jli/libjli.so
Reading symbols from /lib64/libdl.so.2...(no debugging symbols found)...
done.
Loaded symbols for /lib64/libdl.so.2
Reading symbols from /usr/java/jre1.6.0_19/lib/amd64/libverify.so...
(no debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0_19/lib/amd64/libverify.so
Reading symbols from /usr/java/jre1.6.0_19/lib/amd64/libjava.so...(no
debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0_19/lib/amd64/libjava.so
```

```
Reading symbols from /lib64/libnsl.so.1...(no debugging symbols found)...
done.
Loaded symbols for /lib64/libnsl.so.1
Reading symbols from /usr/java/jre1.6.0_19/lib/amd64/native_threads/
libhpi.so...(no debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0_19/lib/amd64/native_threads/libhpi.so
Reading symbols from /lib64/libnss_files.so.2...
(no debugging symbols found)...done.
Loaded symbols for /lib64/libnss_files.so.2
Reading  symbols  from  /usr/java/jre1.6.0_19/lib/amd64/libzip.so...(no
debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0_19/lib/amd64/libzip.so
Reading  symbols  from  /usr/java/jre1.6.0_19/lib/amd64/libnet.so...(no
debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0_19/lib/amd64/libnet.so
Reading  symbols  from  /usr/java/jre1.6.0 19/lib/amd64/librmi.so...(no
debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0_19/lib/amd64/librmi.so
Reading symbols from /usr/java/jre1.6.0_19/lib/amd64/libnio.so...
(no debugging symbols found)...done.
Loaded symbols for /usr/java/jre1.6.0 19/lib/amd64/libnio.so
Reading symbols from /opt/nawg/lib/libheadercodecJNI.so...(no debugging
symbols found)...done.
Loaded symbols for /opt/nawg/lib/libheadercodecJNI.so
Reading symbols from /opt/nawg/lib/libwpss_wsl.so.1...done.
Loaded symbols for /opt/nawg/lib/libwpss_wsl.so.1
Reading symbols from /opt/nawg/lib/libwpss hc.so.2...done.
Loaded symbols for /opt/nawg/lib/libwpss_hc.so.2
Reading symbols from /opt/nawg/lib/libwss_wenc.so...done.
Loaded symbols for /opt/nawg/lib/libwss_wenc.so
Reading symbols from /usr/java/jre1.6.0_19/lib/amd64/libawt.so...done.
Loaded symbols for /usr/java/jre1.6.0 19/lib/amd64/libawt.so
Reading symbols from /usr/java/jre1.6.0_19/lib/amd64/headless/libmawt.
so...done.
Loaded symbols for /usr/java/jre1.6.0_19/lib/amd64/headless/libmawt.so
Reading symbols from /usr/lib/oracle/10.2.0.2/client/lib/libclntsh.so.
10.1...done.
Loaded symbols for /usr/lib/oracle/10.2.0.2/client/lib/libclntsh.so.10.1
Reading symbols from /usr/lib/oracle/10.2.0.2/client/lib/libnnz10.so...
done.
Loaded symbols for /usr/lib/oracle/10.2.0.2/client/lib/libnnz10.so
Reading symbols from /usr/lib/oracle/10.2.0.2/client/lib/libociicus.
so...done.
Loaded symbols for /usr/lib/oracle/10.2.0.2/client/lib/libociicus.so
```

```
Core was generated by 'java -server -DWAPHOME=/opt/nawg/log/wps_var
-DsystemRoot=/opt/nawg -DsystemBin'.
Program terminated with signal 6, Aborted.
[New process 12593]
[New process 22363]
…
[New process 11752]
[New process 11751]

#0  0x00002b08894f4215 in raise () from /lib64/libc.so.6
```

然后执行 bt 命令。

示例 14-4：

```
(gdb) bt

#0  0x00002b08894f4215 in raise () from /lib64/libc.so.6
#1  0x00002b08894f5cc0 in abort () from /lib64/libc.so.6
#2  0x00002b0889df13d7 in os::abort () from /usr/java/jre1.6.0 19/lib/
amd64/server/libjvm.so

#3  0x00002b0889f2a50d in VMError::report_and_die () from /usr/java/jre1.
6.0_19/lib/amd64/server/libjvm.so
#4  0x00002b0889df74c1 in JVM_handle_linux_signal () from /usr/java/jre1.
6.0 19/lib/amd64/server/libjvm.so
#5  0x00002b0889df3cfe in signalHandler () from /usr/java/jre1.6.0_19/lib/
amd64/server/libjvm.so
#6  <signal handler called>
#7  0x00002aaaec5dd0bd in declex (yylval=<value optimized out>, DecParam=
0x572dd840) at lex dec.c:7997
#8  0x00002aaaec5e15c5 in decparse (DecParam=0x572dd840) at yacc_dec.c:1008
#9  0x00002aaaec5d4ff1 in HC_DecodeHeader (Context=0x564b62f0, WspHeader=
<value optimized out>, WspHeaderLength=4096,
 ContentLength=0, HttpHeader=0x5636a390 "B$¨éB$: \r\n", HttpHeaderLength=
0x43caa814) at hc_decoder.c:2517
#10 0x00002aaaec3b2dbe in Java_com_nokia_wap_filter_headercodec_Header-
Codec_cDecode ()
   from /opt/nawg/lib/libheadercodecJNI.so
#11 0x00002aaaab866058 in ?? ()
#12 0x0000000043caa8b0 in ?? ()
#13 0x0000000000000000 in ?? ()
(gdb)
```

从上面 bt 命令获取到的堆栈信息来看，可以清晰地看到问题所在的源代码的行数，即：

```
hc_decoder.c:2517
```

这样就便于分析和定位问题了。

（7）使用 gdb where 输出实例。

示例 14-5：

```
#6 0xfe3c6904 in __1cCosFabort6Fl_v_ () from /landingbj/sharedInstalls/
solaris/wls70sp2/jdk131_06/jre/lib/sparc/server/libjvm.so
#7 0xfe3c59f8 in __1cCosbBhandle_unexpected_exception6FpnGThread_ipCpv_v_
() from /landingbj/sharedInstalls/solaris/wls70sp2/jdk131_06/jre/lib/
sparc/server/libjvm.so
#8 0xfe20a8bc in JVM_handle_solaris_signal () from /landingbj/shared-
Installs/solaris/wls70sp2/jdk131_06/jre/lib/sparc/server/libjvm.so
#9 0xff36b82c in __sighndlr () from /usr/lib/libthread.so.1
#10 <signal handler called> #11 0xe9f90420 in Java_HelloWorld_displayHello-
World ()from /home/usera/wls70/solaris/projectWork/lib/libhello.so
#12 0x90aec in ?? ()
```

（8）使用 gdb thr 输出实例。

示例 14-6：

```
(gdb) thr
[Current thread is 1 (LWP 14 )]
(gdb) info thr
LWP 13 0xff29d194 in  poll () from /usr/lib/libc.so.1
LWP 12 0xff29f008 in _lwp_sema_wait () from /usr/lib/libc.so.1
LWP 11 0xff29f008 in _lwp_sema_wait () from /usr/lib/libc.so.1
LWP 10 0xff29bc2c in _so_accept () from /usr/lib/libc.so.1
LWP 9 0xff29bc2c in _so_accept () from /usr/lib/libc.so.1
LWP 8 0xff29d194 in _poll () from /usr/lib/libc.so.1
LWP 7 0xff29d194 in _poll () from /usr/lib/libc.so.1
LWP 6 0xff29f008 in _lwp_sema_wait () from /usr/lib/libc.so.1
LWP 5 0xff29f008 in _lwp_sema_wait () from /usr/lib/libc.so.1
LWP 4 0xff29f008 in _lwp_sema_wait () from /usr/lib/libc.so.1
LWP 3 0xff29d194 in _poll () from /usr/lib/libc.so.1
LWP 2 0xff29e958 in _signotifywait () from /usr/lib/libc.so.1
LWP 1 0xff29d194 in _poll () from /usr/lib/libc.so.1
LWP 16 0xff29c4fc in door_restart () from /usr/lib/libc.so.1
LWP 15 0xff369774 in private___lwp_cond_wait ()
from /usr/lib/libthread.so.1
* 1 LWP 14 0xff369764 in __sigprocmask ()
from /usr/lib/libthread.so.1
```

14.3.4　探查 HPUX 系统

（1）确认 core 文件由 JVM 产生，即运行$ file <fullpath>/core。

（2）检查 JDK 版本。

（3）使用调试器获得堆栈跟踪和线程信息。

① HPUX 提供了 adb。

② 使用 GNU 的 gdb 也可能会获得有用的信息。

（4）确定导致故障的本地代码类型。

（5）使用 HPUX adb。

确认 JDK 版本代码如下。

示例 14-7：

```
$ java -version                        （获得 JDK 版本）
$ ls /opt/bin/adb  或  which abd       （获得 adb 的位置）
$ export DEBUG_PROG=/opt/bin/adb       （设置为 adb 的位置）
启动 adb：
$adb <path to java command>/java  corefile
输入 adb 命令：
adb> $C          （显示堆栈跟踪的摘要）
adb> $r          （显示寄存器的状态）
adb> $q          （退出 adb）
```

14.3.5 探查 AIX 系统

（1）确认 core 文件由 JVM 产生，也就是运行$ file<fullpath>/core。

（2）检查 JDK 版本。

（3）使用 JVM 进程的 javacore 文件，获得当前线程信息。

（4）确定导致故障的本地代码类型。

（5）使用 AIX javacore 文件。

javacore<WLSpid>.<ID#>.txt 文件示例如下。

示例 14-8：

```
Current Thread Details:
 "ExecuteThread: '10' for queue: 'default'" (TID:0x31c70ad0, sys thread t:
0x3e52df68, state:R, native ID:0xf10) prio=5
at HelloWorld.displayHelloWorld(Native Method)
at servlets.NativeServlet.doGet(NativeServlet.java:85)
at javax.servlet.http.HttpServlet.service
 (HttpServlet.java:740)
at javax.servlet.http.HttpServlet.service
 (HttpServlet.java:853)
at weblogic.servlet.internal.ServletStubImpl$
ServletInvocationAction.run (ServletStubImpl.java:1058)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:401)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:306)
```

14.3.6 探查 Windows 系统

（1）Windows 系统中的 DrWatson 日志文件 drwtsn32.log 与 UNIX 二进制核心文件类似。

（2）该日志文件的产生位置在 c:\Documents and Settings\All Users\Documents\DrWatson。

（3）使用 DrWatson 探查故障原因。

（4）JVM 的 hs_err_pid<WLSpid>.log 文件（如有过）也可能会包含有用的信息。

（5）确定导致故障的本地代码类型。

（6）JVM 日志文件示例。

hs_err_888.log 示例如下。

示例 14-9：

```
An unexpected exception has been detected in native code outside of VM
Unexpected Signal : 11 occurred at PC=0x5a4cf2e4
Function name=Java_HelloWorld_displayHelloWorld
Library=/home/landingbj/user_projects/mydomain/lib/libhello.so
Current Java thread:
at HelloWorld.displayHelloWorld(Native Method)
at servlets.NativeServlet.doGet(NativeServlet.java:85)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:740)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
. . .
Local Time = Wed 17 09:35:39 2004
Elapsed Time = 186
# The exception was detected in native code outside the VM
# Java VM: Java HotSpot(TM) Client VM (1.3.1_06-b01 mixed mode)
```

14.3.7 未提供调试器的系统探查

（1）如果未提供调试器，可能会提供 pstack 和 pmap 实用程序。

（2）实用程序的名称视所用平台而有所不同。

（3）您可能需要下载表 14-1 所列的工具。

表 14-1

平台	pstack	pmap
Solaris	pstack	pmap
AIX 5.2 或更高版本	procstack	procmap
Linux	lsstack	pmap
HPUX	未提供	未提供

（4）使用 Solaris　pstack 命令输出示例如下。

示例 14-10：

```
core 'core' of 20956:/landingbj/sharedInstalls/solaris/wls70sp2/jdk131
06/bin/../bin/sparc/native
----------------- lwp# 14 / thread# 25 -------------ff369764 __sigprocmask
(ff36bf60, 0, 0, e6181d70, ff37e000, 0) + 8
ff35e110 _sigon (e6181d70, ff385930, 6, e6180114, e6181d70, 6) + d0
ff361150 _thrp_kill (0, 19, 6, ff37e000, 19, ff2c0450) + f8
ff24b900 raise (6, 0, 0, ffffffff, ff2c03bc, 4) + 40
ff2358ec abort (ff2bc000, e6180268, 0, fffffff8, 4, e6180289) + 100
fe3c68fc __1cCosFabort6Fl_v_ (1, fe4c8000, 1, e61802e8, 0, e9f90420) + b8
fe3c59f0 __1cCosbBhandle_unexpected_exception6FpnGThread_ipCpv_v_ (ff2c02ac,
fe53895c, fe4dc164, fe470ab4, fe4c8000, e6180308) + 254
fe20a8b4 JVM handle solaris signal (0, 25d5b8, e6180d90, fe4c8000, b,
e6181048) + 8ec
ff36b824 __sighndlr (b, e6181048, e6180d90, fe20a8cc, e6181e14,
e6181e04) + c
ff3684d8 sigacthandler (b, e6181d70, 0, 0, 0, ff37e000) + 708
--- called from signal handler with signal 11 (SIGSEGV) ---
e9f90420 Java_HelloWorld_displayHelloWorld (25d644, e6181224, e61819b8, 0,
2, 0) + 30
00090ae4 ???????? (e6181224, e61819b8, 25d5b8, fe4c8000, 0, 109a0)
0008dc4c ???????? (e61812c4, ffffffff, ffffffff, 97400, 4, e61811b8)
0008dc4c ???????? (e618135c, e61819b8, fe4c8000, 99600, c, e6181250)
0008dc4c ???????? (e61813ec, f76a2f90, e618147c, 99600, c, e61812f8)
0008ddb4 ???????? (e618147c, f68578b8, 0, 99974, c, e6181388)
0008ddd8 ???????? (e618154c, e61815c8, e61815cc, 99974, 4, e6181410)
```

（5）使用 pmap 命令输出示例如下。

示例 14-11：

```
E9500000 1184K read
E9680000 1392K read
E9800000 4608K read
E9F60000 136K read/write/exec
E9F90000 8K read/exec
/home/usera/wls70/solaris/project
Work/lib/libhello.so
E9FA0000 8K read/write/exec /home/usera/wls70/solaris/p
rojectWork/lib/libhello.so
E9FB4000 8K read/write/exec
E9FC0000 120K read/exec /usr/lib/libelf.so.1
E9FEE000 8K read/write/exec /usr/lib/libelf.so.1
...
```

14.4　未生成 core 文件的解决办法

14.4.1　确保服务器发生故障时可以产生 core dump 文件

（1）检查系统和用户对 core 文件大小的限制，也就是运行 ulimit -c。

（2）在 Solaris 系统中，可使用 coreadm 命令确认没有禁用 core dump，也就是执行 coreadm -e process。

（3）在 Linux 系统中，默认情况下会禁用 core dump，也就是检查/etc/security 中的 limits.conf 文件。

（4）在 HPUX 系统中，检查用户进程数据段大小，也就是检查 maxdsiz 值及检查磁盘空间的用户限额。

14.4.2　备用方案：获得最后时刻的 Thread dump

（1）Thread dump 是对 JVM 进程中所有活动线程的原样快照。

（2）捕捉线程在发生故障前的瞬间状态。

服务器启动时启用下列项。

① Sun JVM:　　　-XX:+ShowMessageBoxOnError。

② JRockit JVM:　　-Djrockit.waitonerror。

同时在前台启动服务器。

（3）之后如果 JVM 崩溃，会显示一则提示：Do you want to debug the problem?

（4）回答提示前，您就可以趁机捕捉 JVM 的 Thread dump 了。

14.5　总结：故障排查清单

（1）确认服务器的 JVM 产生了 core 文件，如果没有 core 文件，则设置相关参数，在下次故障发生时捕捉二进制核心文件。

（2）使用二进制核心文件进行探查。

① 使用调试器或其他工具。

② 使用调试器或其他工具。

（3）探查具体的故障原因。

① 删除或替换可疑的本地代码区域。

② 可能需要进行记录和/或详细调试。

（4）继续监视，看是否还会发生故障。

第 15 章　打开文件过多故障

15.1　打开的文件过多概述

WebLogic 在运行过程中会打开创建、打开一些文件，而在操作系统上，允许单个进程打开的文件数是有限制的，当达到操作系统的限制时，就有可能出现故障，比如 WebLogic 不能正常进行 I/O，不能建立 Socket 连接，等等。

15.2　相关知识回顾

15.2.1　在什么时候会打开文件

在下列情况下将打开或创建文件。

（1）JVM 读取服务器及其应用程序所需的大量类时。

（2）建立新的套接字连接（客户端到服务器或服务器到服务器）时，每个套接字（Socket）连接都需要消耗一个文件描述符。

（3）每当新建与服务器的 HTTP 浏览器的连接时，当建立浏览器 HTTP 请求到服务器的连接时，会占用 TCP 套接字。

（4）大容量和大规模应用程序需要同时打开大量文件时。

15.2.2　文件描述符

文件描述符是一个由无符号整数代表的句柄，被进程用来标识一个打开的文件。文件描述符和一个文件对象关联，文件对象中包含类似这样的信息：文件打开的模式，position 类型，初始类型，等等。文件描述符可以继承，并可由子进程使用。操作系统使用固定数量的文件描述符来管理打开的文件。

进程获取文件描述符的最常见的方式如下。

（1）通过本地子程序打开或创建。

（2）从父进程那里继承。

第二种方式允许子进程平等地访问父进程使用的文件。对于每个进程来讲，文件描述符通常都是唯一的。当以 fork 子程序创建子进程时，子进程会获得一份所有其父进程打开的文件描述符的备份。当进程被 fcntl、dup、dup2 子程序复制时，也会发生同样的复制过程。

进程与文件的关系示意图如图 15-1 所示。

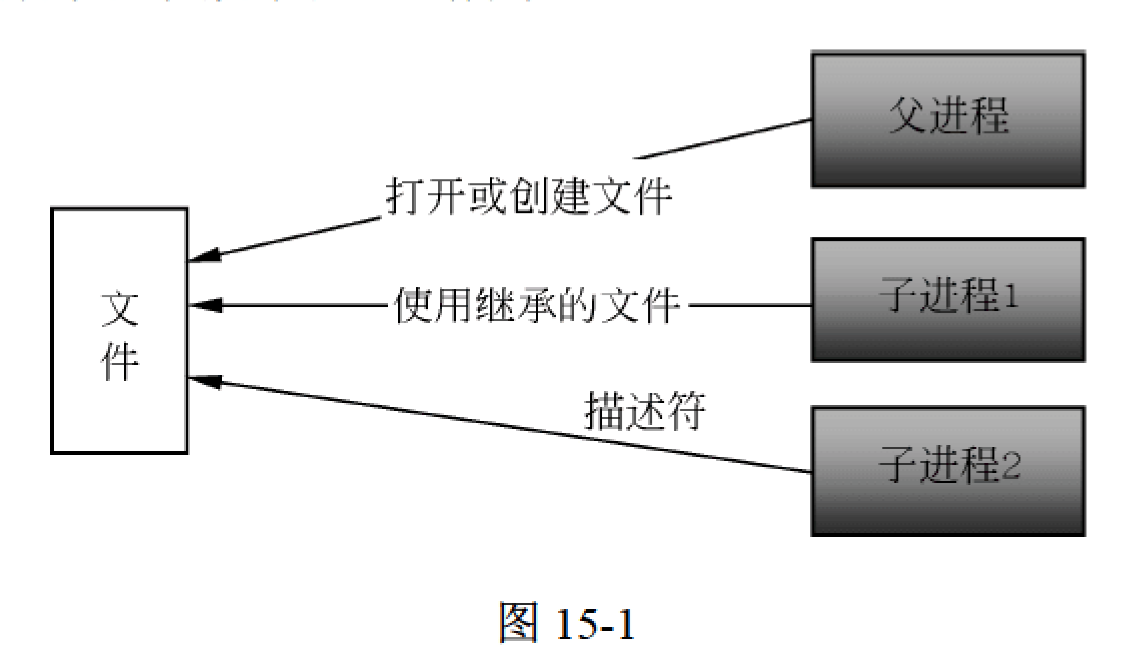


图 15-1

针对每一个进程，操作系统内核维护着一个文件描述符表，所有被进程使用的文件描述符在表里都被进行了索引。

15.2.3　与文件描述符有关的系统参数

文件描述符的个数受操作系统资源限制的控制。

（1）操作系统上可以使用的文件描述符总数。

（2）单个进程最多可以打开的描述符数。

对于不同的操作系统，与文件描述符有关的系统参数不一样，下面将逐一讨论。

1．Windows 系统上的文件描述符

在 Windows 系统上，文件描述符被称为文件句柄，在 Windows 2000 服务器上，打开的文件句柄数限制为 16384。在任务管理器概要里可以对打开的文件句柄数做监控。

2．Solaris 系统上的文件描述符

在 Solaris 操作系统上，/usr/bin/ulimit 工具设定了单个进程可以使用的文件描述符的数量。单个进程可以使用的文件描述符的最大值由 rlim_fd_max 参数指定，默认为 65536。只有 root 用户可以修改这些内存参数。

3．HP-UX 系统上的文件描述符

在 HP-UX 操作系统上，nfile 规定最大可以打开的文件数。nfile 通常由公式((NPROC*2)+1000) 决定，其中 NPROC 通常为((MAXUSERS*5)+64)。例如，假如 MAXUSERS 是 400 的话，那么 nfile 就为 5128。

HP-UX 系统中还有两个内核参数是用来限制单个进程打开文件的最大数的，maxfiles 是单个进程打开文件的软限制，maxfiles_lim 是硬限制。Maxfiles 的值必须不大于 maxfiles_lim 的值

4．Linux 系统上的文件描述符

对于 Linux 操作系统，管理员用户可以在/etc/security/limits.conf 配置文件中设置文件

描述符限制，例如：

```
soft nofile 1024
hard nofile 4096
```

对于系统范围内的描述符限制可以这样设置，在/etc/rc.d/rc.local 脚本中加入：

```
echo 4096 > /proc/sys/fs/file-max
echo 16384 > /proc/sys/fs/inode-max
```

其中的 1024、4096 和 16384 等具体的数值可以根据需要更改。

5．AIX 系统上的文件描述符

对于 AIX 操作系统，文件描述符限制在/etc/security/limits 文件中设定，默认是 2000。可以使用 ulimit 命令或 setrlimit 子程序进行更改。最大值由 OPEN_MAX 规定。

15.2.4 文件描述符的释放

（1）文件描述符是在文件关闭或进程终止时被释放的。

（2）要想重用某个文件描述符，必须关闭与之关联的所有文件描述符（父进程/子进程）。如果 close()系统调用没有传回失败代码，那么相关联的文件描述符在后续就可以被 open()系统调用，并在分配描述符时使用。

（3）打开新文件时会重用关闭的文件描述符。

（4）TIME_WAIT 结束时才会释放 TCP 套接字文件描述符。

关闭的 Socket 会过渡到 TIME_WAIT 状态，以确保在连接期间所有的数据都被传送了，最终的 ACK 确认标志结束数据的传输。这个周期也会延缓分配给套接字的文件描述符的释放。

在 UNIX 操作系统上，TIME_WAIT 的持续时间由内核参数 tcp_time_wait_interval 规定。

在 Windows 系统上，这个周期由注册表里的 TcpTimedWaitDelay 规定，具体位于 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters.里。

在 UNIX/Windows 系统上这个等待时间的默认值都是 240 秒。

15.3 打开的文件过多问题及故障

15.3.1 与打开的文件过多有关的问题

（1）如果没有正确关闭文件，文件描述符可能未被释放。

（2）如果子进程使用的描述符未关闭文件，将不能重用该文件描述符。

（3）TIME_WAIT 结束前，TCP 套接字会使文件描述符保持打开状态。

（4）不要依赖 GC 和对象清除功能来释放文件描述符。

不应该依赖垃圾回收和对象清除去释放非 Java 资源，例如文件描述符。当遇到错误时，应该使用 close()调用，并处理它的输出。

（5）只有固定数量的文件描述符可供打开的文件使用。

15.3.2　与打开的文件过多有关的故障症状

1．进程报：I/O Exception 错误

进程在执行文件 I/O 时达到了文件描述符极限，并影响 I/O 操作时，就会报此异常，如下所示。

示例 15-1：

```
Java.io.FileNotFoundExceptin:/***/wlserver_10.3/server/lib/consoleapp/w
ebapp/images/sort_up.gif(Too many open files)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java:112)
at weblogic.utils.classloaders.FileSource.getInputStream(FileSource.
java:31)
at weblogic.servlet.internal.warSource.getInputStream(warSource.java:65)
at weblogic.servlet.FileServlet.sendFile(FileServlet.java:400)
Truncated.see log file for complete stacktrace
```

2．进程报：Socket Exception 错误

进程打开的文件过多时，在打开新套接字连接时，可能就达到了文件描述符极限，如果错误影响到底层的 TCP 协议，就会报 Socket Exception 错误。

示例 15-2：

```
Java.net.SocketException:Too many open files
at java.net.PlainSocketImpl.accept(Compiled Code)
at java.net ServerSocket.implAccept(Compiled Code)
at java.net.ServerSocket.accept(Compiled Code)
at weblogic.t3.srvr.ListenThread.run(Compiled Code)
```

15.4　打开的文件数过多问题探查

当问题发生时，收集系统上当前文件的使用信息。如果可能的话，收集预期文件的使用信息，将实际的数据与预期的数据做对比，然后做相应的更改。另外也可以使用特定平台工具以确定系统的当前设置信息。

15.4.1　服务器日志文件描述符极限

服务器在启动时会记录一条信息，说明操作系统上设置的文件描述符的硬极限，如下示例。

示例 15-3：

```
<ListenThread> <system> <> <000000> <Using effective file descriptor limit
of: '8192' open sockets/files.>
```

15.4.2　类 UNIX 平台上探查

1. 类 UNIX 平台上的 lsof 工具

在类 UNIX 平台上，可以使用 lsof（LiSt Open Files）工具查看有关打开的文件的个数和文件描述符的具体信息，包括文件类型、大小及 inode 等信息。lsof 还可以显示 Socket 连接的类型（TCP/UDP）、监听的地址、端口、连接状态等信息。通过使用 lsof 等工具，在出现异常时，可以使用此命令查看 Java 进程是否达到了可以打开的文件的最大上限，也可以查看应该关闭的文件是否仍然处于 open 的状态。如果发现了应该关闭的文件，那么您就需要分析一下为什么该关闭的文件没有关闭。

lsof 的语法信息可以通过执行 lsof -h 查看。对于特定的进程，lsof 的语法是这样的：lsof -p <进程的 PID>，lsof 执行结果示例如下。

WebLogic 线程（线程号：6345）打开的文件数。

```
lsof -p 6345 |wc -l
```

WebLogic 线程（线程号：6345）打开的文件。

示例 15-4：

```
[root@yuchao ~]# lsof -p 6345 | more
COMMAND    PID      USER    FD   TYPE  DEVICE   SIZE      NODE   NAME
Java  6345  weblogic  cmd    DIR 8.1 4096    820574
/home/weblogic/Oracle/Middleware/user projects/domains/yx domain
Java  6345  weblogic  r t d  DIR 8.1 4096    2    /
Java  6345  weblogic  t x t  REG 8.1 47308   590310
/home/weblogic/Oracle/Middleware/jdk160_11/bin/java
Java  6345  weblogic  mem   REG   8.1  112212  835586  /lib/ld-2.3.4.so
Java  6345  weblogic  mem   REG   8.1  1547588 835599  /lib/tls/libc-2.3.4.so
Java  6345  weblogic  mem   REG   8.1  16748   835614  /lib/libdl-2.3.4.so
Java  6345  weblogic  mem   REG   8.1  214060  835612  /lib/tls/libm-2.3.4.so
Java  6345  weblogic  mem   REG   8.1  107928  835618  /lib/libresolv-2.3.4.so
Java  6345  weblogic  mem   REG   8.1  81140   835652  /lib/libresolv-2.3.4.so
```

（1）losf 工具可以在 Solaris、Tru64、HP-UX、Linux、AIX 等系统上使用。lsof 的最新版本可以从如下链接获得：http://ftp.cerias.purdue.edu/pub/tools/unix/sysutils/lsof/。

（2）在 HP 系统上，也可以使用 glance 工具，glance 工具可以用来分析当系统上运行 WebLogic 时总的打开文件的数量。glance 工具可以从 http://www.hp.com 上获得。

（3）如果在您的类 UNIX 平台上不能使用 lsof 工具，您也可以查看/proc/<pid>/fd 目录，这个目录下有某个进程的所有文件描述符的信息。

15.4.3　Windows 平台上探查

1．handle

在 Windows NT 或 Windows 2000 系统上可以使用命令行工具 handle，它会报告打开的文件的句柄信息。handle 句柄的主要语法为 handle -p <process name> ，如下所示：

```
D:\Tools>handle -p java
```

handle 工具可以从下列链接中获得：

http://download.sysinternals.com/Files/Handle.zip

2．process explorer

Windows 上的另外一个工具是进程探测器：process explorer，可以用于监控文件句柄。它有图形用户界面，可以显示更多关于每个正在运行的进程信息。可以使用该工具搜索某个特定的句柄。process explorer 示例如图 15-2 所示。

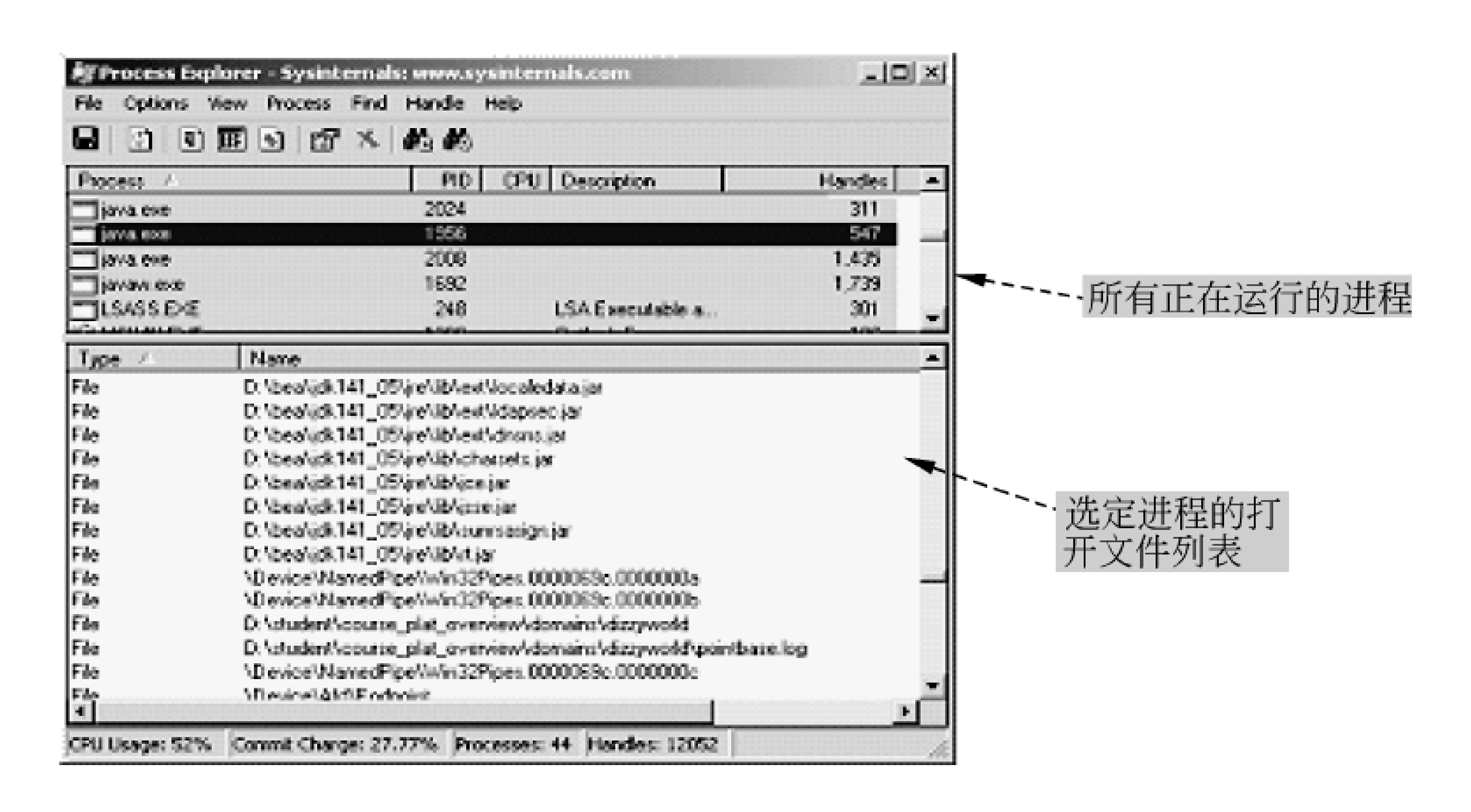


图 15-2

process explorer 工具，可以从如下链接获取：

http://www.sysinternals.com/ntw2k/freeware/procexp.shtml

15.5 故障排除策略

15.5.1 故障排除策略一

（1）对于引发异常的进程，需要定期检查该进程打开的文件和/或连接。

（2）将检查结果与进程的预期描述符要求进行比较。

（3）还要将检查结果与操作系统对进程和总体的文件描述符限制进行比较。

（4）根据需要提高 OS 限制来弥补文件描述符的不足，然后继续进行进程监视。

（5）重复上面的步骤。

15.5.2 故障排除策略二

（1）如果文件描述符使用量继续随时间推移而增加，那么检查是否存在下列情况。

① 正在按预期方式关闭文件。如果存在未按预期关闭的文件的话，查看为何未被正常关闭。

② 正在创建不必要的文件。比如某个驱动库，不断地为每个新建的 JDBC 连接载入文件。

③ 各个类以 jar 形式统一加载，而不是各自单独加载。

jar 文件载入也会降低使用的文件描述符的数量。一个描述符可以被每个 jar 文件使用，但是一个文件描述符将会被单独载入的每个类使用。

④ 有大量处于 TIME_WAIT 状态的连接。

（2）更正所有应用程序文件处理故障。

（3）根据需要将 TIME_WAIT 周期缩短为一个切合实际的值。

TIME_WAIT 周期是某个连接在被关闭以前处于 TIME_WAIT 状态的时间长度。在繁忙的服务器上，默认时间是 240 秒，TIME_WAIT 周期会延迟其他的连接尝试，因此会减慢连接相关的文件描述符的释放，也会限制最大的连接数。

（4）继续监视打开的文件和/或连接。

第 16 章　内存不足和内存泄漏故障

16.1　内存不足/内存泄漏错误概述

16.1.1　内存不足/内存泄漏简介

内存不足（Out Of Memory，OOM），是指出现了没有空闲内存可供 JVM 或本地代码用于分配新对象或内存块的情况。内存不足错误的直接结果就是没有空闲内存供 JVM 或本地代码使用。

内存泄漏是指已经分配好的内存或对象，当不再需要时没有得到释放，这样就不能再继续提供给别的程序使用了。当泄漏的内存占总的可用内存的比例达到一定程度后，可用的空闲内存会越来越少，就会出现内存不足现象。

因为内存泄漏会导致内存不足问题出现，所以两者的探查方法完全相同。

16.1.2　内存不足分类

内存不足基本可以分为如下 3 类。

（1）Java 堆内存不足。

（2）本地内存不足。

（3）WTC（Weblogic Tuxedo Connector）内存不足。

16.2　关键知识点回顾

16.2.1　Java 堆

Java 堆是 JVM 用于分配 Java 对象的内存，Java 堆中的对象又包括存活的和不可用对象。JVM 是支持 WLS 实例的进程，它负责维护 Java 堆，如果从运行着的程序中的任何指针都不能到达某个对象的话，JVM 就将该对象断定为“垃圾”。JVM 要求有足够的空闲堆空间以创建新对象及执行相关的操作。JVM 通过垃圾回收释放堆里未使用的 Java 对象。

Java 堆的最大值通常是服务器启动时，使用 Java 命令中的-Xmx 标志来设定的。服务器启动时，JVM 将根据设置的 Java 堆最小、最大值来预分配一定数目的内存。如果没有手工指定 Java 堆的最大值，JVM 会根据服务器实际内存的总大小及当时可用空闲内存的大小等因素后决定采用的值。但是，建议您还是手动设定一下最大值。

16.2.2 本地内存

本地内存是JVM用于其内部操作的内存。JNI代码和第三方本地模块（例如，本地JDBC驱动程序）也可以使用本地内存。

JVM 使用的本地内存的大小取决于生成的代码量、创建的线程数、垃圾回收时用到的内存、生成代码及优化时用到的临时内存等因素。如果使用了第三方的本地模块，第三方本地模块也会使用本地内存，比如本地 JDBC 驱动会分配本地内存。

最大本地内存大小取决于操作系统虚拟进程大小限制和 Java 堆内存最大值。例如，在 32bit 的操作系统上，通常应用程序可以分配的内存总量为 3GB 左右；如果 Java 堆的最大值是 1GB，那么本地内存可能的最大值就约为 2GB 了。

16.2.3 进程大小

进程大小是 Java 堆、本地内存与加载的可执行文件和库所占用的所有内存的总和，进程大小受操作系统的限制。关于进程内存进一步的信息，可以参考 16.2.6 小节。

16.2.4 垃圾回收

GC 会自动检测和释放不再使用的堆内存。Java 运行时，系统会执行 GC，这样程序员不再需要显示释放对象。通常在空闲内存降低到某一水平或内存分配达到某一数量后会自动触发垃圾回收。

16.2.5 可及对象及对象的可及程度

程序可以直接到达的对象以及程序以其能够直接到达的对象为起点，沿着指针引用链最终可以到达的对象都称为可及对象。重要对象的集合，也就是为人们所知的 roots，被假设成可以到达，例如，机器寄存器、栈、指令指针、全局变量等，这些对象是直接可及的。可及对象的可及程度从最强到最弱，又可细分如下几种类型。

（1）强可及：线程可以不用遍历任何引用对象就可以到达对象。新创建的对象对于创建它的线程来讲是强可及的。

（2）软可及：做不到强可及，但是可以通过遍历一个软引用到达。

（3）弱可及：非强可及也非软可及，但是可以通过遍历一个弱引用到达。当到某个弱可及对象的弱引用被清除以后，该对象就可以被清除了。

（4）虚可及：非强可及、非软可及、非弱可及，已经被清除了，但是某些虚引用还在引用它。

（5）不可及：不能通过上面的任何一种方式到达该对象。该对象符合被回收的标准。

16.2.6 虚拟内存与物理内存

计算机的可用物理内存总量是 RAM 和交换空间之和，由计算机中的所有进程共享。

在任何特定的时间，物理服务器（计算机）上所有正在运行的进程使用的虚拟内存的总和不能超过计算机的物理内存。

在进程范围内，内存地址是虚拟的。每个进程都会获得属于自己的地址空间。在 32 位的操作系统上，这个地址空间的范围是 0～4GB。这跟物理服务器上可用的 RAM 及交换空间没有关系。操作系统内核会将这个虚拟地址映射成物理地址。物理地址指向物理内存的某个位置。进程的虚拟地址空间受操作系统最大进程大小的限制，在 32 位的操作系统上，操作系统内核会为自己保留 4GB 虚拟地址空间的一部分。一般大约保留 1～2 GB，剩下留给应用程序使用。对于 Windows 操作系统，默认情况下 2GB 给应用程序用，2GB 保留给内核使用。但是，在某些 Windows 变种中，有一个/3GB 的开关可以用来改变上面提到的 2GB/2GB 的比率，以使应用程序可以获得 3GB 的内存。

物理内存和虚拟内存的示例如图 16-1 所示。

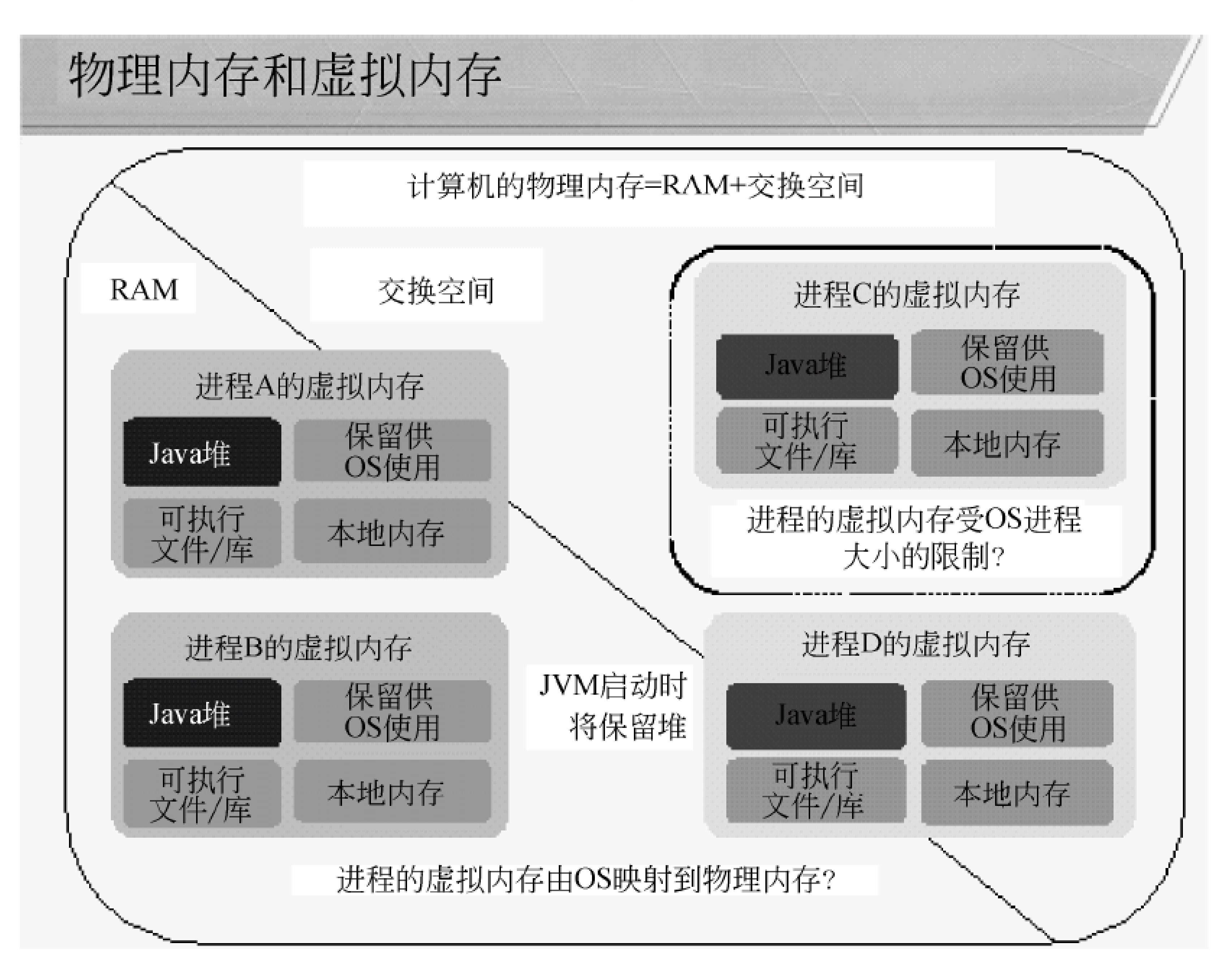


图 16-1

16.2.7　WTC：WebLogic Tuxedo Connector

WTC（WebLogic Tuxedo Connector）提供了 WLS 应用程序和 Tuxedo 服务器之间的互操作能力，它允许：① WLS 客户端调用 Tuxedo 服务；② Tuxedo 客户端生成调用 WLS EJB 方法的服务请求。

16.3　内存不足错误分类探讨

内存不足分为 Java 堆内存不足，本地内存不足，WTC 内存不足，下面将会首先讨论

前两种，WTC 内存不足将在 16.3.3 节专门讨论。

16.3.1 Java 堆内存不足错误

当 JVM 没有足够的堆空间可用于分配新的 Java 对象时，就会出现 Java 堆内存不足错误。

1. Java 堆内存不足故障症状

（1）JVM 会抛出 Java.lang.OutOfMemoryError 异常。

（2）stdout 或 stderr 中将显示一则消息。

（3）应用程序可以忽略错误或处理异常，例如友好退出。

（4）如果应用程序忽略错误，线程将会退出（且将不会出现在 Thread dump 中）。

（5）WebLogic Server 的执行线程只是通过记录状态来处理异常。

（6）但连续发生的错误将导致健康监控线程关闭服务器。

在遇到 Java 堆内存不足错误时，JVM 会在抛出 Java 内存不足错误以后，让应用程序决定如何处理。应用程序可能会处理这个错误，并决定以一种安全的方式关闭自己，或者继续运行忽略该错误。如果应用程序不处理这个错误的话，那么报错的线程将会退出，并且在 Java Thread dump 里不会看到该线程（线程应该出现在 Thread dump 里才对）。对于 WLS，如果是执行线程抛出的这个错误，那么 WLS 会处理它并会记录到日志里。如果继续报这个错误的话，核心健康监控进程将会关闭服务器。

2. Java 堆内存不足成因

Java 堆内存不足的可能原因包括以下几方面。

（1）没有足够的堆。

Java 堆内存不足的根本原因可能是设置的 Heap 堆最大值过小，不足以满足应用的处理要求。比如，因额外的客户端负载或对应用程序处理方式的更改而导致的 Java 堆内存不足。

（2）对象的活动时间过长。

如果应用程序中存在很多长时间运行的对象（如 HTTP 会话），那么可能的话，尝试降低对象的存活时间。比如，调整 HTTP 会话的超时时间，这有助于更快地回收空闲会话对象。

（3）应用程序在内存中缓存对象。

应用程序可能会在内存里缓存 Java 对象，cache 大小可能会慢慢的增大，但是不应该无限制增大，应该受限于对 cache 里对象数量的限制。建议在数据缓存中尽量使用 Java 软引用，因为当 JVM 用完 Java 堆时，软可及对象可以保证会被移除掉。

（4）内存泄漏。

例如，在将连接传回连接池后没有关闭 JDBC 结果集对象。当在应用程序服务器上使用 JDBC 连接池时，在最终的 block 里必须显示关闭 JDBC 语句和结果集对象。对连接池的连接对象调用 close()的话，仅是将连接传回了连接池以达到重新使用的目的，但是并没

有关闭连接及相关联的语句或结果集。

（5）因存在 JVM 错误，未能执行完整的 GC。

如果在内存不足错误出现以前并没有执行完整的垃圾回收，那么 JVM bug 可能是造成内存不足的原因。

（6）内存碎片。

Java 对象需要连续的内存，在需要大内存块时，只有小的内存碎片是可以使用的话，JVM 就不能分配大的对象，也会报内存不足错误。内存压缩（Compact）是垃圾回收的一部分，并且通常只有在（明确）要求的情况下才会进行。

3. 探查 Java 堆内部不足错误

（1）探查概述。

对于 Java 堆内存不足错误，可采取以下措施。

① 确保启用了 verbose GC。

也就是在启动服务器时使用了 java –verbose:gc 开关。对于 JRockit ，使用-verbose 取代-verbose:gc，因为这会将代码生成信息也追加到 GC 信息中去。Verbose 输出会在 Java 进程的 stdout/stderr 上打印 GC 的活动信息，所以建议将 Java 进程的 stdout/stderr 输出重定向到一个文件中去。

② 探查内存不足错误是否发生在运行了完整的 GC 之后。

③ 与 JVM 供应商联系，确认完整的垃圾回收可以删除软可及、弱可及和虚可及对象。

④ 确定所使用的 GC 方案的类型，进而确定 GC 输出中应出现哪些 GC 消息。

⑤ 检查是否执行了内存压缩以减少内存碎片。

⑥ 还要留意初始（和周期）JVM 堆可用性/占用率。

使用管理控制台，可以定期的监控可用的堆空间的量，可以这样操作：依次单击 Servers→myServer→Monitoring→Performance 按钮。

（2）完整 GC 的输出。

① 可以检查在 OOM 信息之前是否进行了完整的垃圾回收。当完成完整垃圾回收之后，会打印类似如下所示的信息。

示例 16-1：

```
[memory ] 7.160: GC 131072K->130052K (131072K) in 1057.359 ms
```

消息格式为[memory] <start>: GC <before>K-><after>K　(<heap>K), <total>ms。其中：

<start>　GC 的开始时间（秒），从 JVM 启动开始计算。

<before> 回收前对象所使用的内存（KB）。

<after>　回收后对象所使用的内存（KB）。

<heap>　回收后的堆大小（KB）。

<total>　执行回收的总时间（毫秒）。

> **注 意**
>
> 不同厂商和版本的 JVM 输出信息的格式并不完全一致，所以需要参考 JVM 的帮助信息，以正确地理解格式。

② 即使使用 verbose 信息，也没有方法断定软可及、弱可及、虚可及对象是否确实被移除掉了。如果在发生 OOM 时，您怀疑这些对象还仍然存在的话，那么需要利用 Heap dump 工具做进一步的分析，或者联系您的 JVM 供应商。

（3）分析 GC 输出。

分析 GC 输出以确定 GC 是不是按照预期在进行工作。GC 输出可以反映以下情况。

① OOM 错误是否发生在运行完整 GC 之后。

完整垃圾回收可以回收不可及、虚可及、弱可及、软可及对象所占用的空间。

② GC 返回了多少空闲空间。

③ 内存使用量是否增加缓慢（即表明发生了内存泄漏）。

④ 是否进行了内存压缩

内存压缩通常可以消除可能存在的碎片问题。确保 JVM 会做恰当的内存压缩工作，这可以防止在分配大对象时触发 Java OOM 错误。压缩工作包括将对象（数据）从 Java 堆里的一个地方移动到另一个地方，并更新对移动的对象引用，使其指向新的位置。除非有必要，否则，JVM 不会压缩所有的对象。这是为了减少 GC 周期的暂停时间。

⑤ 进一步确认是不是因为内存碎片导致的内存不足错误。

我们可以分析详细的垃圾回收信息，以进一步确定是不是因为碎片造成的内存不足。如果在即使还有空闲的 Java 堆内存的情况下，还是出现 OOM 错误，那么就很有可能是因为内存碎片引起的内存不足。如下所示，在执行 GC 后，仍然存在碎片。

显示在执行 GC 后仍存在碎片的 GC 消息示例。

示例 16-2：

```
[memory ] 8.162: GC 73043K->72989K (131072K) in 12.938 ms
[memory ] 8.172: GC 72989K->72905K (131072K) in 12.000 ms
[memory ] 8.182: GC 72905K->72580K (131072K) in 13.509 ms
. . .
java.lang.OutOfMemoryError
```

从以上信息，已看到指定的最大堆大小为 128MB，在实际使用的内存只有 70MB 左右时，在还有 45%左右空闲 Java 堆内存的情况下，JVM 就抛出了 OOM 错误。因此能很可能是碎片问题导致的 OOM。这可能是 JVM 的 bug 或限制，您应该联系您的 JVM 供应商，以做进一步的确认。

（4）针对 Java 堆内存不足的应用程序分析。

如果 JVM 正确执行了 GC，则应该是应用程序问题所致。

① 如果应用程序使用缓存对象，那么可采取以下措施。

a. 确保对缓存对象数量施加了限制。

b. 考虑降低缓存限值来减小总体堆大小。

c. 使用 Java 软引用来确保 JVM 用完堆空间时缓存对象会被删除。因为在 JVM 用完 Java 堆后，软可及对象会确保被移除。

② 如果应用程序使用了活动时间长的对象，可减少这些对象的数量或缩短它们的生命周期，例如，缩短 HTTP 会话的超时期间，这有助于更快地回收空闲会话对象。

③ 查找内存泄漏，例如，不正确的连接池处理。

④ 如果内存增长和负载增加或添加应用程序相关，则有可能是正常的内存需求增长。

（5）进一步分析 Java 堆 OOM。

① 如果迄今为止所进行的分析都不起作用，可使用 JVM 事件探查器。

a. 该探查器将实现 JVM 事件探查器接口（JVM Profiler Interface，JVMPI），如 Jprobe 或 OptimizeIt。

b. 确定占用堆的对象。

c. 确定对象在代码中的创建位置。

d. 有关其用法的详细信息，可参考相应的事件探查器文档。

② JVM 事件探查器通常需要较高的系统开销，因此建议一般不要在生产环境中使用。

③ 与应用程序团队合作，查明可能存在的内存泄漏或改进对象的使用和/或生命周期状况。

Sun 公司的 JVMPI（JVM 事件探查器接口）是一个用于探查的实验性接口。它是一个 JVM 和进程中探查器代理之间的双向功能调用接口。一方面 JVM 会将不同事件的信息通知给探查器代理，比如与堆分配、方法调用等相应的事件。另一方面，探查器代理会通过 JVMPI 发出控制请求，比如 JVMPI 可以基于前端探查器的需要，打开/关闭针对某个特定事件的通知。所以，如果上面的所有建议都不适合于您的应用程序的话，那么就需要使用基于 JVMPI 的探查器了，比如 Jprobe 或 OptimizeIt 等，以找出哪些对象在占用 Java 堆。探查器也会给出对象是从 Java 代码里的哪个地方创建的等相关的细节信息。

参考特定探查器的问题，以理解在使用探查器的情况下如何设置及启动应用程序。通常，基于 JVMPI 的探查器会有很高的开销，并且会明显降低应用程序的性能，因此，不建议在生产环境中使用。更多关于 Jprobe 和 OptimizeIt 的信息，可以参考：

http://www.borland.com/optimizeit

http://www.quest.com/jprobe

（6）JRockit 功能。

① Java 运行时分析器 Java Runtime Analyzer。

Java 运行时分析器（Java Runtime Analyzer，JRA），是 WebLogic JRockit 开发项目组用来分析 JRockit 运行时，及运行在 JRockit 上的应用程序的性能的内部工具。它提供了丰富的 JRockit 内部信息，这些信息对开发人员来讲很有用。JRA 包括以下两个部分。

a. 收集有关 JVM 和当前正在运行的 Java 应用程序的数据。这一部分运行在 JVM 内部，负责收集信息，并将其保存到某个文件中。

b. 使用分析器工具来查看收集到的信息。记录文件会被 jra 分析工具打开，jra 分析工具是一个常规 Java 应用程序，可以用来对记录文件做处理，以使记录文件里的信息更容易为用户所查看。

② JRockit 8.1 及之上版本支持 Java 运行时分析器（jra）。

a. 该分析器对 JRockit JVM 及 JRockit 上运行的 Java 应用程序都能够进行运行时性能分析。

b. 使用该分析器来收集 GC 数、热方法及软、弱、虚引用的数目等信息。

③ 花几分钟时间来做一个记录，当 JVM 出现性能问题或挂起时，可以用来分析数据。

16.3.2 本地内存不足错误

本地内存中，当 JVM、JNI 代码或本地库无法为对象或内部操作分配空间时，就会出现本地内存不足错误

1. 本地内存不足故障症状

如果本地内存用尽，那么将出现下列故障。

（1）将抛出本地内存异常错误。

（2）stdout 或 stderr 中将显示一则消息。

（3）JVM 的通常做法是处理内存状态，记录消息，然后退出。

（4）如果 JVM 或任何其他已加载的模块（例如 libc 或第三方模块）不处理异常，操作系统将通过 sigabort 强制 JVM 退出。

（5）通常会产生二进制核心转储文件（core dump）。

JVM 如果不能再获得更多的本地内存时就会报本地内存不足错误。这通常会在进程达到了操作系统的进程大小限制或计算机用完了 RAM 和交换空间后出现。当出现这种情况时，JVM 会进行处理，并记录一个消息，说明它用完了本地内存或者不能够再获取内存了，然后退出。如果 JVM 或任何其他载入的模块（如 libc 或第三方模块）不处理这种情况，也就是说忽略这种情况的话，操作系统会给 JVM 发送 sigabort 信号使 JVM 退出。通常，JVM 接收到 sigabort 信号以后，会产生一个二进制核心文件。

2. 本地内存不足成因

本地内存不足的可能原因包括以下几方面。

（1）进程的虚拟内存大小（RAM 和交换空间）达到了操作的进程大小极限。

（2）物理内存（RAM 和交换空间）不足以满足计算机中所有线程的需要。

如果物理服务器没有足够的 RAM 和交换空间的话，那么操作系统就无法再为进程提供更多的内存，就会造成内存不足错误。物理服务器上必须要有足够的 RAM 和交换空间以满足所有进程运行的需要。

（3）分配给 JVM 堆的内存过多。

JVM 在所有类都被载入、所有方法都被调用（代码生成已经完成）以后，使用的本地内存量就会趋于平稳。对于大多数应用程序来讲，这通常能在程序启动时最开始的几个小时里完成。完成之后，JVM 使用的新增本地内存就会很少，仅仅会在运行时类加载及代码生成（因为优化）时需要本地内存。

所以，如果使用的 Java 堆大小小于最大 Java 堆大小，那么可以考虑减小最大 Java 堆大小，以给 JVM 更多的本地内存。由于操作系统限制了进程的大小，而最大 Java 堆大小和本地内存大小是一种此消彼长的关系，您需要在 Java 堆大小和本地内存之间达到一个平衡。

（4）本地库或 JNI 代码中发生内存泄漏。

检查一下您是否在使用任何第三方的本地模块，比如数据库驱动。这些本地模块也可能会分配本地内存，从而就有可能会导致内存泄漏的情况出现。另外，检查一下应用程序

是否在使用某些 JNI 代码，这也可能会造成本地内存泄漏。

（5）内存碎片。

与 Java 堆内存不足类似，也就是在需要大内存块时，只有小的内存碎片可以使用.

3．探查本地内存不足

（1）对于本地内存 OOM 错误的探查方法如下。

① 采用的探查方法与对 Java 堆 OOM 采用的探查方法相同。

a. 通过-verbose:gc 开关收集 GC 信息。

b. 确认 GC 的运行合乎预期，例如是在 OOM 发生前运行。

c. 留意 JVM 堆的初始（和周期）可用性、占用率。

② 定期监视进程的虚拟内存大小。

a. 在 UNIX/Linux 上，使用 ps -p <PID> -o vsz。

b. 在 Windows 系统上，按照下面的步骤监控虚拟进程的大小：

执行“开始”→“运行”命令，然后在“运行”文本框中输入“perfmon”，最后按 Enter 键。在弹出来的窗口中，单击中上部的+标志，如图 16-2 所示。

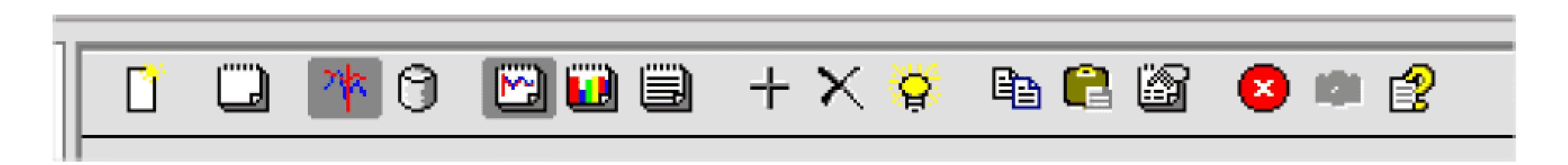


图 16-2

在弹出来的窗口中：

性能对象：选择：Process（非默认的 Processor）。

计数器：选择：Virtual Bytes。

范例：选择 JVM（Java）。

接着单击“添加”按钮，最后单击“确定”按钮。

③ 确定是否使用了任何本地模块或 JNI 代码。

④ 还要检查计算机的物理内存总量（RAM 和交换空间之和）是否足以满足所有正在运行的进程的需要。

在启动 WLS 时，加上-verbose:gc，以收集垃圾回收相关的信息。另外，检查 JVM 实际使用的 GC 模式，以确认 GC 是不是按照预期在进行。另外，使用 Admin Console，以定期监控可用的堆空间的量，可以执行 Servers→myServer→Monitoring→Performance 命令。

自应用程序启动起，就定期记录进程的虚拟内存大小，直到 JVM 出现 OOM 问题。确认进程是不是真的达到了操作系统的大小限制。另外，检查是否使用了任何第三方本地模块，比如 JDBC 驱动。它们也可能分配本地内存，同样也有可能导致内存泄漏问题。

（2）影响本地内存的因素。

影响进程的虚拟内存大小的因素如下。

① 是进程运行时所占用的地址空间。

② 受操作系统进程大小值的限制。

a. 32 位 UNIX：进程大小限值为 4GB，保留 1～2 GB 供 OS 自己使用。

b. Red Hat Linux AS 2.1：可供应用程序使用的进程大小为 3GB。

c. Windows：进程大小限值为 2GB（默认值），但可以增加到 3GB。

③ 还由 Java 堆共享，JVM 在服务器启动时会保留指定的最大值。

④ 还可能由 JNI 代码或本地模块（例如本地 JDBC 驱动程序）进行分配。

⑤ 还会受计算机物理内存及计算机中运行的其他进程的限制。

虚拟进程的大小是 Java 堆、本地内存及由载入的可执行文件占用的内存的和。在 32 位的操作系统上，进程的最大虚拟地址空间是 4GB，这其中，内核为自己做了保留，一般为 1～2GB，剩下的供应用程序使用。

在 Windows 操作系统上，默认内核会保留 2GB，剩下的 2GB 给应用程序使用，但是某些 Windows 的变种版本中，有一个/3GB 的开关，可以用来改变上面提到的 2GB/2GB 的比率，可以让应用程序使用 3GB 的空间。

对于 Red Hat Linux AS 2.1 操作系统，应用程序可以使用 3GB。

对于其他操作系统，需要参考相应的官方文档。所以，最大可用本地内存受限于操作系统虚拟进程大小限制和最大 Java 堆内存大小（通过-Xmx 标志指定）。例如，假如应用程序分配了 3GB，Java 堆的最大大小是 1GB，那么最大可用本地内存就是 2GB。需要注意的是，最大可用堆内存，在 JVM 启动时，就已经被保留了，不能用做其他目的。

4．本地内存 OOM 解决方法

（1）使用收集到的探查数据来解决 OOM 错误。

（2）如果怀疑发生了内存泄漏，可集中精力查找泄漏源。

① 第三方模块（例如 JDBC 驱动程序）或 JNI 代码可能会发生泄漏。

② 在可能的情况下尝试替换纯 Java 实现，以确认泄漏源。

如果本地内存大小不断增大，那可能在本地代码中存在内存泄漏。那么，应该首先定位在哪里出现了泄漏。检查您是否使用了第三方的本地模块，比如数据库驱动。这些模块也会分配本地内存，这样就有造成泄漏的可能。您可以使用纯 Java 驱动代理本地数据驱动。另外也检查一下应用程序是否使用了 JNI 代码。这也可能会造成内存泄漏。如果可能的话，尝试在不使用 JNI 代码的情况下运行应用程序。

（3）如果存在内存泄漏，其他以增加可用的本地内存大小为手段的解决办法仅能够延缓故障发生，并无其他作用。

（4）如果使用的堆大小远低于最大堆大小，考虑减少最大堆大小，以提供更多的本地内存。

如果已使用的 Java 堆大小在最大可用堆大小的范围内，那么可以降低最大堆大小，以为 JVM 提供更多的本地内存，供 JVM 处理其他东西。这不是解决内存泄漏的方法，只是一个可以尝试的避开内存泄漏的方法。因为操作系统限制了进程的大小，我们需要在 Java 堆大小和本地内存大小之间取得平衡。

（5）如果 RAM 和交换空间不足，则需要升级计算机或将工作分配给其他计算机。

（6）JVM 使用本地内存。

① 加载类和生成代码，但在启动几小时后，内存使用量通常会稳定下来。

② 可能会发生运行时类加载和代码优化。

③ 禁用运行时代码优化。

a. 如果使用的是 JRockit，可使用-Xnoopt 标志。

b. 如果使用的是 Sun 的 Hotspot，可使用-Xint 标志在解释模式中运行 JVM（不会出现代码生成）。

（7）最后，如果无法查明本地内存 OOM 错误的成因，可采取下列方法。

① 与 JVM 供应商联系，找到跟踪本地内存分配调用的方法。

② 与第三方模块或 JNI 代码供应商联系，了解是否可以启用调试/跟踪功能来消除成因。

（8）继续收集和分析有关 OOM 错误发生时间和发生原因的信息。

（9）如果存在多个成因，缩小探查范围可能需要一些时间。

如果在做了上面的这些步骤后，还是无法定位哪里使用了本地内存，那么就需要联系 JVM 供应商了，以获取特殊的方法来追踪本地内存分配调用，并得到更多关于内存泄漏的信息。

16.3.3　WTC 及 WTC 内存不足问题分析

1．WTC 简介

WTC 的具体特点如下。

（1）提供了一种在 WLS 应用程序和 Tuxedo 服务之间的互操作功能。

（2）允许 WLS 客户端调用 Tuxedo 服务。

（3）允许 Tuxedo 客户端像调用 Tuxedo 服务那样调用部署在 WebLogic Server 上的 EJB。

（4）可以 WLS 应用程序集成到已经存在的 Tuxedo 环境中。

Tuxedo 和 WLS 相关调用示例如图 16-3 所示。

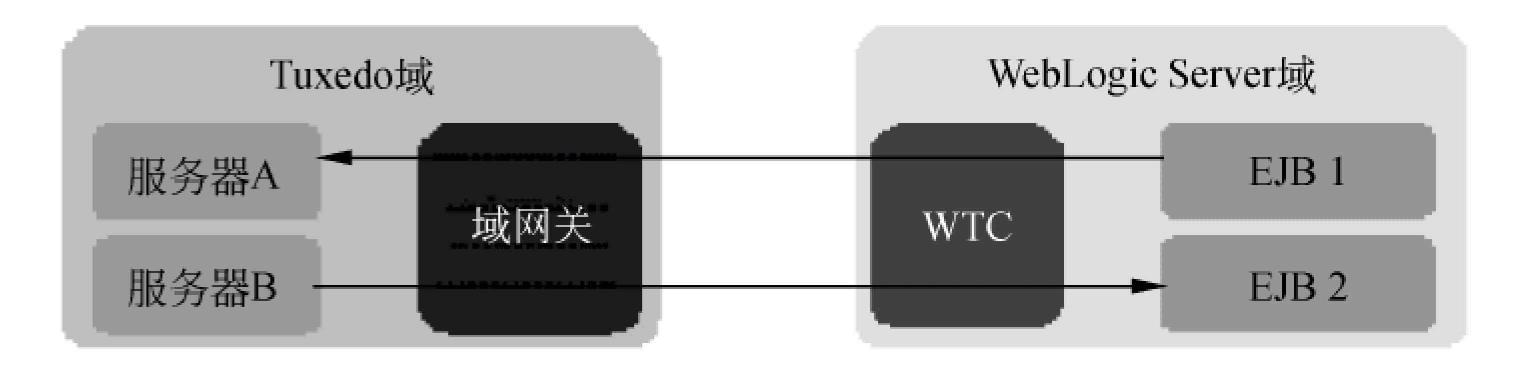


图 16-3

2．WTC 内存不足故障症状

WTC 在出现内存不足问题时，通常会抛出以下异常：java.lang.OutOfMemoryError。

3．WTC 内存不足成因

造成 WTC 内存不足的原因，基本可以归为如下几类。

（1）WLS/Tuxedo 配置不正确。

WTC 内存不足最常见的成因是 WLS 端的配置不正确、Tuxedo 端的配置不正确或两者

配置均不正确。有时候是因为某个 bug 也会造成 WTC 内存不足。

备注：与端口冲突有关的几个问题在 WLS 8.1 中已得到解决。

（2）OS 中设置的进程的最大线程数不足。

如果您遇到了下面的信息：

```
java.lang.OutOfMemoryError:unable to create new native thread
```

那么是因为 UNIX 内核对单个进程里最大线程数的设置太小了。需要将这个限制增大到足够大，以容纳 Java 应用程序里的所有线程，再加上虚拟机自身的少数线程。

（3）堆大小不足或计算机物理内存不足也可能是问题所在，可参考以上各小节。

检查一下是否给 WebLogic Server 分配了足够的堆及系统上有足够的内存。如果 WLS 在启动时仅仅设置成了 128MB，那么就将其改大一点，以确保不是因为堆内存设置的过小造成的问题。

（4）分配给永久生成区（Perm）的 JVM 堆空间不足。

永久生成区是用来保持 VM 自身的 reflective 对象的，比如类对象和方法对象。这些 reflective 对象是直接分配到永久生成区里去的，并且它的大小的设置独立于其他的区。通常，可以忽略对永久生成区的大小的设置，因为默认的大小就足够了。然而，如果程序使用了太多的类，或者应用程序大量动态生成/加载类，则可能会需要更大的永久生成区。

4．WTC 内存不足错误探查

（1）检查 WLS 和 Tuxedo 配置是否正确。

① 网络地址必须唯一且匹配正确。

a. WebLogic Server 监听地址应是独立的，如 7001 端口。核实 WLS 和 Tuxedo 没有使用相同的端口号，这在过去造成过 OOM 错误。

b. WTC Local Access Point 与 Tuxedo Remote Domain 匹配。

c. WTC Remote Access Point 与 Tuxedo Local Domain 匹配。

② WTC 服务必须拥有自己特有的一组本地地址和远程地址。

关于①和②两种情况，可以参考如图 16-4 所示的 WLS 域和 Tuxedo 域示例。

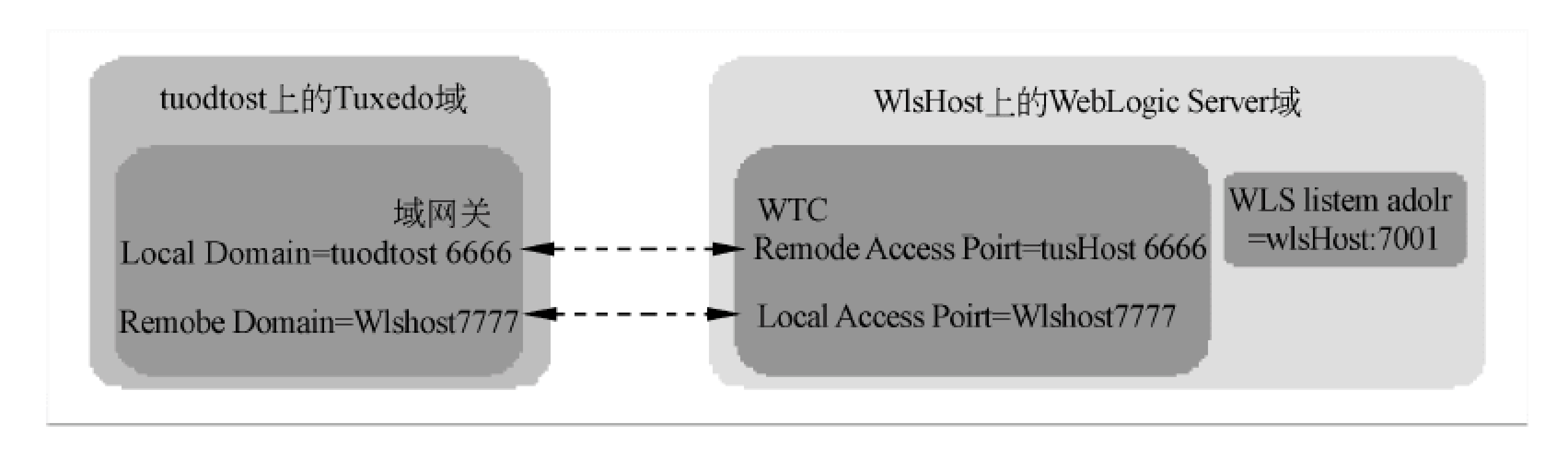


图 16-4

③ 当配置 WTC 时，参考下面的指导原则。

a. 在配置里，至少要有一个本地 Tuxedo 访问指针。

b. 在配置里，可能会不止一个 WTC 服务。

c. 不能将同一个 Server 作为两个 WTC 服务的 target。一个 Server 只能是某一个 WTC

服务的 target。

d. 在选定 target server 以后，任何在 WTC 服务里对配置的改变，在 target server 实例里都不会被更新。如果想要更新的话，只能是将 WTC 服务从 target server 实例里删除，然后将更新以后的 WTC 服务添加到 target server 实例里去。

（2）检查 Tuxedo 端。

① 与 WTC 实例化的每个连接都必须有独立的域。

如果一个已经存在的 Tuxedo 应用程序已经使用了 Tuxedo 域，那么就必须要在域配置文件里添加一个新域，以完成为每一个到 WTC 实例的连接服务。如果已经存在的 Tuxedo 应用程序没有使用域，那么域服务器必须要添加到应用程序的 TUXCONFIG 里去。另外，也必须要创建一个与 WTC 实例相应的、带 Tuxedo/TDomain 条目的 DMCONFIG。

② 确保启用了编码，即本地域或远程域的 MTYPE 设置应为不同或不设置（NULL）。

WTC 要求 Tuxedo 域的 encoding 一直处于打开状态，也就是说要求 DMCONFIG 文件里 MTYPE 的值要么未设置，要么设置成了 NULL。MTYPE 的值用来对域做分组，以使对域之间消息的编/解码可以分路。但是对于从 WLS 到 Tuxedo（或者反过来），DMCONFIG 文件里 MTYPE 值应该设置成 NULL 或者不设置，因为是从 Java（WLS）到 C（Tuxedo），这样编码/解码一直都是必要的。如果想要知道为什么，简单地想一下字符串，在 Java 是一个字符占两个字节，而在 C 里，则是一个字符占一个字节。如果 MTYPE 字段的值和域配置文件中 DM_LOCAL_DOMAINS 和 DM_REMOTE_DOMAINS 部分都一样的话，数据的编码/解码就可以分路。MTYPE 的值可以是长度最长为 15 个字符的任意字符串。它仅仅是用来做对比的。

备注：如果 MTYPE 的值没有指定的话，默认就是将编码/解码设置成了 on。

如果设置了 MTYPE 的话，您将会遇到下面的报错。

示例 16-3：

```
<Error> <Posix Performance Pack><Uncaught Throwable in processSockets
java.lang.OutOfMemoryError <<no stack trace available>>
```

如果您遇到了下面的错误信息的话：

```
java.lang.OutOfMemoryError: unable to create new native thread
```

那么说明，UNIX 操作系统内核设定的单个进程的最大线程数(max_thread_proc)太小了。需要将这个值增大到足以容纳 Java 应用程序里所有的线程+虚拟机自身的少量线程。

（3）确保在 Java 堆内分配的空间不小于 256 MB。

如果要加载大量类，就需要确保 Java 堆内分配的空间大小不小于 256MB。可以通过 Java 命令中的 MaxPermSize 参数设置该值。

（4）启用 WTC 的调试功能。

要收集更多信息，可以启用 WTC 的调试功能。调试输出可用于分析问题，也可作为支持案例发送给 BEA 技术支持部门。

① WLS 6.1 启用 WTC 调试功能。

对于 WLS 6.1，在 WTC 启动类里设置：

```
BDMCONFIG=path_to_my_bdmconfig.xml_file,TraceLevel=100000
```

然后重启 WLS。

示例 16-4：

```
<StartupClass
   Arguments="BDMCONFIG=C:\path\bdmconfig.xml,TraceLevel=100000"
   ClassName="weblogic.wtc.gwt.WTCStartup" Name="WTCStartup"
   Targets="examplesServer"/>
```

② WLS 7.0 和 8.1 启用 WTC 调试功能。

更改启动脚本中的 JAVA_OPTIONS：

```
JAVA_OPTIONS=-Dweblogic.wtc.TraceLevel=100000
```

③ 通过管理控制台启用调试模式。

选择 myServer>Logging>Server/General 选项卡，然后选中 Debug to Stdout 复选框，将 stdout 严重性阈值设置为 info。

注：不管以哪种方式启用调试模式，都需要重新启动 WebLogic Server。

④ WTC 调试输出示例。

示例 16-5：

```
####<Aug 12, 2003 1:12:22 PM EDT> <Debug> <WTC> <SPOZ> <examplesServer>
<Thread-9> <kernel identity> <> <180046> <]/dsession(0)/tpcommit/50/
weblogic.wtc.jatmi.Txid@60e9fef3>
####<Aug 12, 2003 1:12:22 PM EDT> <Debug> <WTC> <SPOZ> <examplesServer>
<Thread-9> <kernel identity> <> <180046> <[/TuxXidRply/get_specific_reply/
    weblogic.wtc.gwt.gwdsession@0/true>
```

大约 2 分钟后，示例如下。

示例 16-6：

```
####<Aug 12, 2003 1:14:24 PM EDT> <Warning> <JTA> <SPOZ> <examplesServer>
<Thread-0> <kernel identity> <> <110030> <XA resource [OatmialResource] has
not responded in the last 120 second(s).>
```

16.4　故障排除检查清单

16.4.1　故障排除检查清单综述

（1）确定 OOM 状态的发生位置。

① 是在 Java 堆还是在内存本地。

② 抑或与 WebLogic Tuxedo Connector 有关。

（2）收集和分析 Verbose GC 输出。

① 检查 GC 的执行时机和结果是否合乎预期。

② 查找内存泄漏和内存碎片问题。

（3）请记住以下两点。

① 如果有内存碎片，则表明 GC 有问题。

② 如果内存泄漏是成因，则增加可用内存大小除了可以延缓故障发生外，不会起到其他作用。

16.4.2　Java 堆内存不足故障排除检查清单

（1）如果 GC 执行正确，可探查应用程序与内存有关的处理。

① 缓存。

② 活动时间长的对象。

③ 增大负载或添加应用程序只会使堆大小的要求不断提高。

（2）要进行更详细的探查，可尝试使用下列方法。

① 用 JVM 事件探查器，以了解占用堆的对象。

② 用 JRockit JRA 功能来分析 JVM 自身及 Java 应用程序的运行时性能。

16.4.3　本地内存不足故障排除检查清单

（1）检查计算机的物理内存量及 OS 的进程虚拟内存大小限制。

（2）定期监视进程的虚拟内存大小。

（3）如果物理内存不足，可升级计算机或将工作分配给其他计算机。

（4）在可能的情况下考虑减小最大 JVM 堆大小，以提供更多本地内存。

（5）确定是否使用了本地模块或 JNI 代码。

（6）必要时请与 JVM 供应商或其他供应商联系，以启用调试/跟踪功能来确定内存的使用情况。

16.4.4　WTC 内存不足故障排除检查清单

（1）第一步是确认配置，特别要确认网路地址是否唯一。

（2）检查分配给永久生成的 JVM 堆空间是否不足。

（3）启用 WLS 调试功能，查找任何异常的处理。

第 17 章　不可恢复堆栈溢出故障

17.1　什么情况下可导致堆栈溢出

应用程序递归过深时会发生堆栈溢出。应用程序可能会发生一般 StackOverFlow 异常。不可恢复的堆栈溢出是指应用程序得到 an irrecoverable stack overflow has occurred. Unexpected Signal 11(SEGV)消息时发生的堆栈溢出。

堆栈溢出示例输出如下。

示例 17-1：

```
Unexpected Signal : 11 occurred at PC=0xfb9c22ec
Function name=write (compiled Java code)
Library=(N/A)
Current Java thread:
Dynamic libraries:
0x10000 /opt/bea/jdk131/jre/bin/../bin/sparc/native_threads/java
0xff350000 /usr/lib/lwp/libthread.so.1. . .
# HotSpot Virtual Machine Error : 11
# Error ID : 4F530E43505002BD 01
#
# Please report this error at
# http://java.sun.com/cgi-bin/bugreport.cgi
# Java VM: Java HotSpot(TM) Client VM (1.3.1_07-b02 mixed mode)
```

如果递归程度超过线程的堆栈大小，递归方法调用就会引发堆栈溢出。下列项目中存在递归错误可引发堆栈溢出：应用代码、应用服务器代码和 JVM 自身的代码；另外某些已知 JVM 错误也可能会导致堆栈溢出。

17.2　堆栈溢出的故障症状

日志或控制台通常会报告 StackOverflowError。不可恢复的堆栈溢出会使 JVM 崩溃，并会试图产生二进制核心文件。通常情况下：① 二进制核心文件可以说明，同一应用程序代码函数被反复多次调用。② 但它不一定能准确定位到哪个具体的应用代码函数。

17.3　堆栈溢出探查

17.3.1　确定可以利用的信息

（1）查看日志中的堆栈跟踪。
（2）如果产生了二进制核心文件则探查该文件。
（3）收集 Thread dump 信息，探查具体的故障原因。
（4）这项工作可能要求探查者熟悉应用程序代码。
（5）应用程序调试记录可能会有帮助。
（6）是否为已知 JVM 问题？

17.3.2　查看日志中的堆栈跟踪

StackOverflowError 可能会在服务器日志或控制台中显示程序调用堆栈的反向跟踪。遗憾的是，很少有提供堆栈反向跟踪的情况，因此需要采用其他诊断方法。因发生不可恢复的堆栈溢出而崩溃的 JVM 可能会产生 JVM 日志文件，其中包含有可能导致了该二进制核心文件的库的详细信息。

JVM 的日志文件如下。

（1）位于服务器的启动目录中。

（2）名为 hs_err_pid<WLSpid>.log，其中 <WLSpid> 是服务器进程的进程 ID。

参考实例文件如下。

示例 17-2：

```
An unexpected exception has been detected in native code outside the VM.
Unexpected Signal : 11 occurred at PC=0x5a4cf2e4
Function name=Java_HelloWorld_displayHelloWorld
Library=/home/laningbj/user_projects/mydomain/lib/libhello.so
Current Java thread:
at HelloWorld.displayHelloWorld(Native Method)
at servlets.NativeServlet.doGet(NativeServlet.java:85)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:740)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853). . .
Local Time = Wed 17 09:35:39 2004
Elapsed Time = 176
# The exception was detected in native code outside the VM
# Java VM: Java HotSpot(TM) Client VM (1.3.1_06-b01 mixed mode)
```

17.3.3　探查二进制核心文件

如果 StackOverflowError 导致 JVM 崩溃，正常情况下会产生 core 文件。可以查看二进制核心文件，了解 JVM 终止是在哪一刻发生的。在某些情况下，不会产生 core 文件。

更多细节内容，请参见第 14 章服务器 core dump 分析。

（1）WLS 可能会正常关闭，因而不会产生 core 文件。

（2）如果 JVM 不崩溃，则不会产生 core 文件。

（3）进程或文件大小限制可能会阻止产生 core 文件。

（4）可以采取措施来确保能够产生 core 文件。core 文件往往是探查服务器故障原因所必需的，要确保服务器发生故障时可以产生了 core 文件。

（5）检查 core 文件的系统和用户大小限制，也就是运行 ulimit -c。

（6）在 Solaris 系统中，检查/etc/system 中的 core 文件转储大小设置，也就是检查 set sys:coredumpsize=0。

（7）在 Linux 系统中，默认情况下会禁用 core 文件转储，也就是检查/etc/security 中的 limits.conf 文件。

（8）在 HPUX 系统中，检查用户进程数据段大小，也就是检查 maxdsiz 值。

（9）使用 quota-v 命令检查磁盘空间的用户限额。

17.4　堆栈溢出的解决办法

首先确定堆栈溢出问题的成因，您可能需要采取以下措施。

（1）修改 JVM 参数增加 JVM 的线程堆栈大小。每个 Java 线程有两个堆栈，一个用于 Java 代码，一个用于 C 代码，JVM 的参数-Xss 控制 Java 代码的堆栈大小，如果错误的递归调用是问题所在，可以考虑适当调高-Xss 的值，不过这种方法并不能避免堆栈溢出，而只能延缓它的发生。

（2）纠正应用程序代码中的错误，确保递归算法不会导致死循环并可以返回结果。如果怀疑应用程序代码有问题，利用以下代码片段可能有助于查明问题所在。

示例 17-3：

```
catch ( StackOverflowError e ) {
          System.err.println("Exception: " + e );
          // Here is the important thing to do
          // when catching StackOverflowErrors:
          e.printStackTrace();
          // do some cleanup and destroy the thread or unravel if possible.
    }
```

查看最近对应用程序代码所做的所有更改，看其中是否有递归调用。在有可疑代码的地方添加调试语句。jsp_error 页中有以下标记将导致无限递归：

<%@ page errorPage="jsp_error"%>

如果错误处理代码内出现异常，可删除此标记并输出堆栈跟踪。这样可以在错误页面中找到问题。解决此问题之后，就可以查看发送到此页面上的原始错误。WebLogic JSP 表单验证标记可导致递归错误。确保 <wl:form> 的确未将 action 属性设置为包含该 <wl:form> 的页面，这样会导致无限循环，结果将导致堆栈溢出错误。

（3）检查已知 JVM 问题。

使用对象数组的数组时，可能会发生堆栈溢出错误。在 java.util 包（JVM v1.3.1 和 1.4.1）中使用 getProperty() 时，也可能会发生堆栈溢出错误。

不要执行以下代码行：

示例 17-4

```
Properties p = new Properties(System.getProperties());
```

而执行以下代码。

示例 17-5：

```
Properties p = new Properties();
p = System.getProperties();
```

17.5　故障排除检查清单

探查日志文件，了解出错位置。查看崩溃的服务器产生的 core 文件。

如果没有 core 文件，则设置相关参数，在下次故障发生时捕捉 core 文件。

进行 Thread dump，找到递归方法调用。

向可疑应用程序代码添加调试跟踪。

检查应用程序基础结构代码。

（1）jsp_error 页，login/auth/error 部分代码。

（2）WebLogic JSP 表单验证标记。

检查已知 JVM 问题。

第 18 章　缓存满异常故障

18.1　实体 bean 池加载和缓存加载

18.1.1　实体 bean 概述

实体 Enterprise Java Bean（EJB）是持久性数据（如“客户” 或者是“银行账号”等数据库实体）在内存中的标示。

（1）允许大量客户端访问实体，每次更改后的更新都将持久保存。

（2）使用主键来标识。

（3）可以加载到池中或缓存中以提高性能。

18.1.2　实体 bean 池加载和缓存加载

WebLogic Server 提供了下列项来增强实体 bean 的性能。

一个容纳了匿名 bean 的自由池（Free Pool）用于调用 finder 方法、home 方法和创建实体 bean。

包含具有标识（即主键）或当前参与了事物的实例的缓存，如图 18-1 所示。



图 18-1

18.1.3　实体 bean 的生命周期

实体 bean 在调用 business 方法时会从自由池转到缓存；会保持 ACTIVE 状态，直到客户端的事务被提交或者是回滚；会保持 READY 状态，除非其他 bean 需要空间；将以匿名 bean 的形式返回到自由池中，为新加载到缓存的 bean 让出空间，如图 18-2 所示。

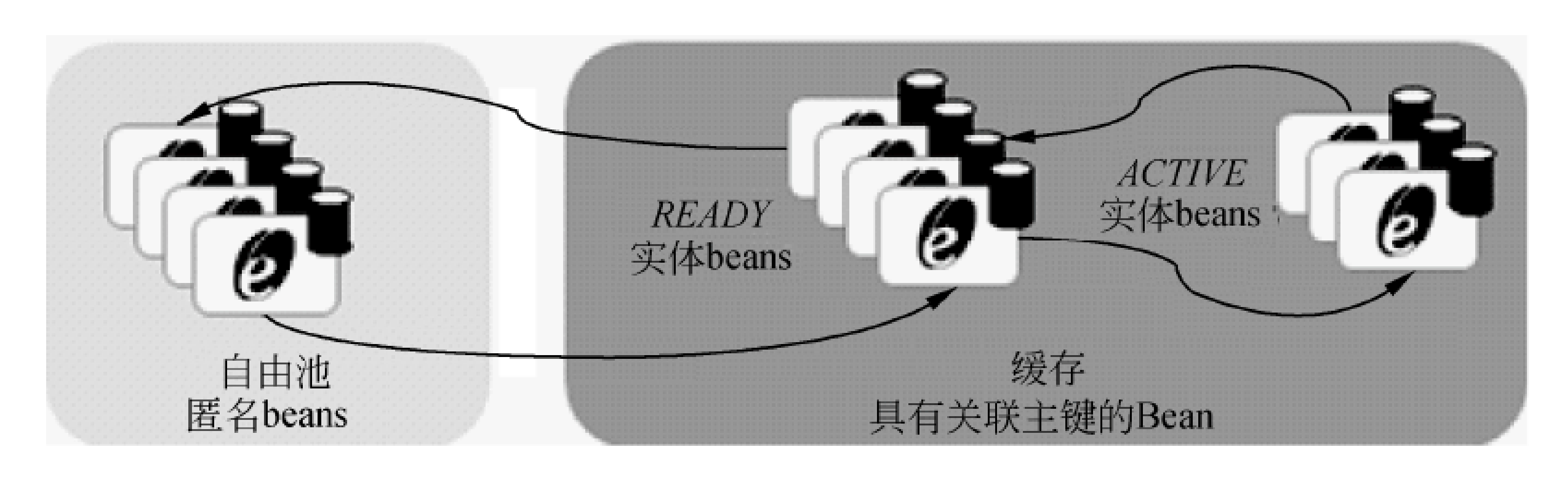


图 18-2

18.1.4 实体 bean 池和缓存大小

设置下列参数可以控制实体 Bean 自由池（free pool）的大小。

① 服务器启动时创建的匿名实例数 initial-beans-in-free-pool（默认是 1000）。

② 自由池中允许的最大实例数 max-beans-in-free-pool。

对于缓存，设置一下参数可控制缓存中允许的最大 READY bean 和 ACTIVE bean 的数量：max-beans-in-cache。

18.2 有状态会话 bean 缓存加载

18.2.1 有状态会话 bean 回顾

有状态会话 EJB 代表特定客户端维护状态；创建于客户端使用 Home 接口之时；在客户端连接被删除或 bean 被删除前将一直供客户端专用。

例如，在线购物车就是一个有状态会话 bean，它维护客户作为购买而准备挑选的物品。

18.2.2 有状态会话 bean 的生命周期

在客户端使用 Home 接口实例化 bean 时，创建并加载到缓存中，因为和 Entity bean 不同，没有自由池的存在。最后调用的方法完成时即变为空闲。可以钝化到磁盘，以释放缓存。客户端会话关闭时将被删除。一旦钝化（或者是空闲），即可能在“超时”时被删除，如图 18-3 所示。

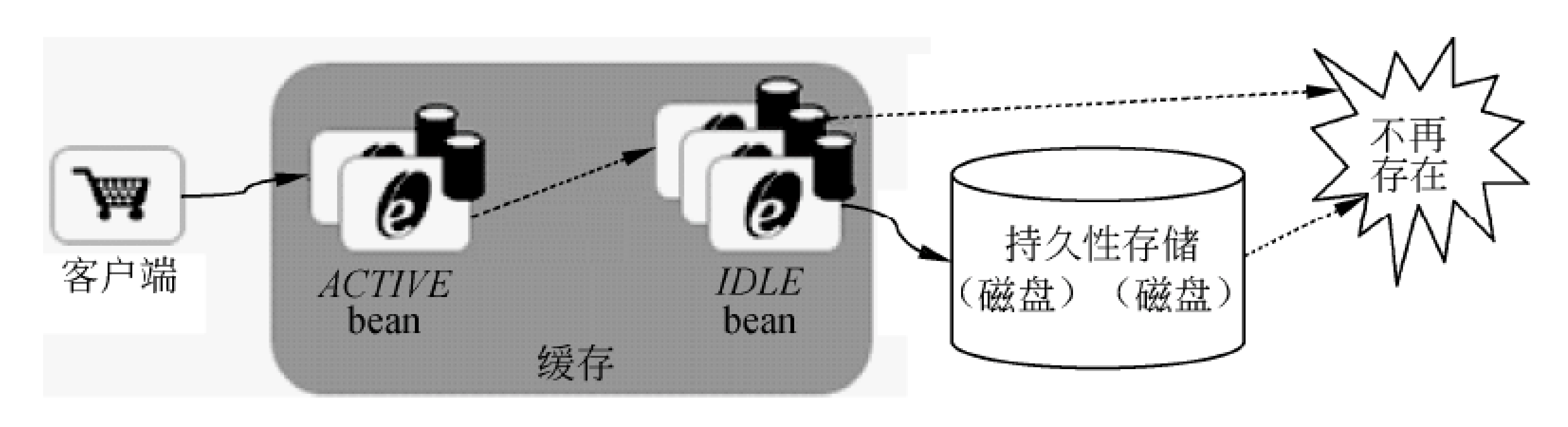


图 18-3

18.2.3 有状态会话 bean 参数

设置下列参数可控制有状态会话 bean 缓存加载。

（1）max-beans-in-cache：最大缓存大小。

（2）idle-timeout-seconds：空闲 bean 和钝化 bean 超时期间。

（3）cache-type：钝化是松散还是紧密。

18.2.4 EJB 缓存和 JVM 堆

（1）在默认情况下，每个有状态会话 bean 或实体 bean 类型。

① 使用单独的缓存。

② 每个都有已定义的最大值。

（2）JVM 堆。

① 用于存储所需的缓存。

② 必须大到足够容纳所有缓存的大小。

18.3 缓存满问题的故障症状和成因

（1）缓存满问题的故障症状包括下列内容。

① 达到缓存中允许的最大 bean 数时抛出 CacheFullException。

② bean 实例未充满缓存时，抛出 CacheFullException 或 NullPointerException。

③ 如果堆由于缓存的 bean 而耗尽，可能会发生 OutOfMemoryError。

（2）缓存满问题的成因包括下列内容。

① 缓存中允许的最大 bean 数设置得不够大。

② 实体 bean 不符合从缓存中删除的条件，即其状态未从 ACTIVE 恢复为 READY。

③ 相对于 JVM 堆空间大小，自由池和缓存中允许的最大 bean 数过大。

18.4 探查缓存满问题

18.4.1 精确定位缓存满问题

需要了解系统内使用的 bean 类型和它们的配置情况。

可能在特定的 bean 的缓存中发生：① 这通常可以缩小问题的发生范围；② 但可能是由其他缓存中的问题（如内存不足）引起的，需要一个能对其他缓存产生影响的解决方案！

也可能在若干个缓存中发生：① 如果是 bean 类型相似或使用相似的缓存参数，则将问题分类，这可能是较适当的做法；② 如果 bean 类型不相似或缓存的配置各不相同，则

可能必须分别探查每个问题。

18.4.2　探查缓存满问题

1．监视 EJB 池和缓存

可以使用 WebLogic 管理控制台监视自由池和缓存的当前状态。

① 执行 deployments→EJBs→Entity 命令。

② 单击 customize this table 按钮以显示更多的信息。

可以使用管理控制台监视自由池和缓存的当前状态，如图 18-4 所示。

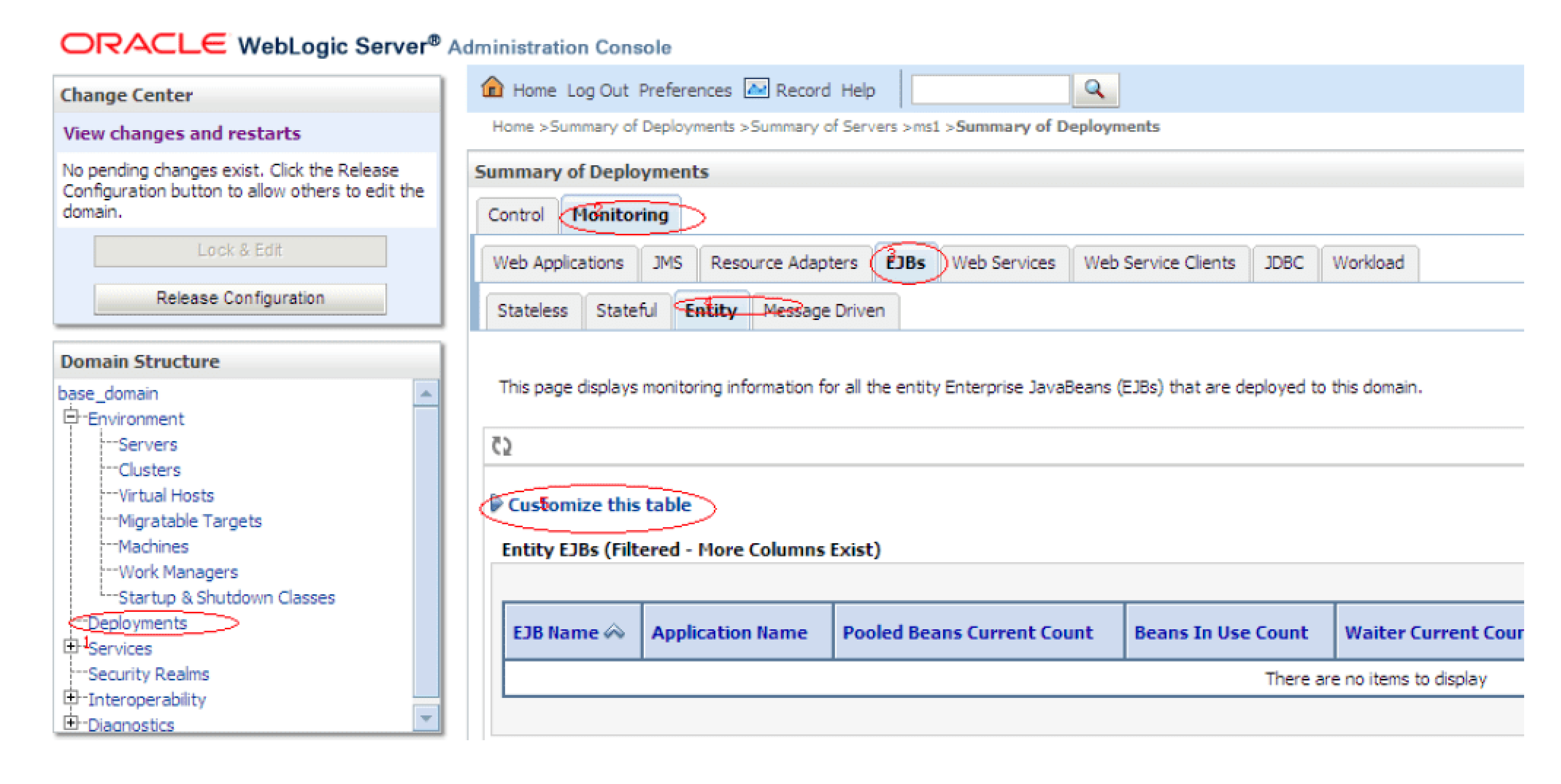


图 18-4

2．探查缓存中的最大 bean 数

（1）监视缓存中的 bean 数：定期进行，以监视日常使用量；在高峰期间进行，已记录最大使用量；在关键操作期间进行，这些操作会对应用程序产生显著影响。

（2）如果缓存中允许的最大 bean 数不足，可增加该值并衡量增加后的效果。

18.4.3　缓存满问题的成因

如果仅仅增加最大缓存大小并不能够解决问题，请考虑是否是下列已知问题所导致的。

对于实体 bean，实体 bean 的主键类的实现可能不正确。

bean 可能由 finder 方法提前或意外加载入缓存，关联缓存加载也会导致这种情况。

对于有状态会话 bean，应用程序可能未有效地删除不再需要的 bean。超时前 bean 在缓存中持续的空闲时间可能过长。

实体 bean 主键缓存索引由事务加主键构成。为识别和匹配加载到缓存中的 bean，主键类提供了 equals()和 hashcode()方法。

为查找某个实体 bean，将对加载到当前缓存中的 bean 进行测试。

- 找到匹配项，加载到缓存中的 bean 将用于客户端请求。
- 未找到匹配项，将创建一个新的 bean 实例，使用目标主键加载到缓存中，并将其用于客户端请求。

如果主键类方法 hashcode()和 equals()的实现不正确，将找不到匹配目标键的加载到缓存中的 bean，加载到缓存中的 bean 实例未得到重用，但也未被删除。

确保 hashcode()方法为每个可能的主键值返回一个唯一的值。

检查 equals()方法的以下方面：

（1）比较正确的字段与多字段键。

（2）对值为 null 的键字段的处理。

示例不正确主键类代码如下。

示例 18-1：

```
public boolean equals(Object sh){
  boolean result = false;
  if (sh instanceof SalesAcct) {
    SalesAcct key2 = (SalesAcct)sh;
    if ((this.prodID != null
          && this.prodID.equals(key2.getProdID()))
          && (this.AcctNo != null
          && this.AcctNo.equals(key2.getAcctNo()))) {
      result = true;
    }
  }
  return result;
}
```

1．容器管理的持久性

（1）容器管理的持久性（Container Managed Persistence，CMP）处理运行时底层数据对象的持久性，包括① 数据库记录的创建和删除；② Business 方法调用间的更新。

（2）为数据对象生成方法。

① 以插入新数据库记录或删除现有数据库记录。

② 以按主键查找记录并将记录实例化为实体 bean。

③ 以供基于查询语言（Query Language，QL）的其他 find 或 select 方法使用。

2．实体 bean 的 CMP

（1）使用 CMP 时，实体 bean 可能会在下列情况下被加载到缓存中。

① finder 方法返回对 bean 的引用时。

② 调用 bean 上的第一个方法时。

③ 这两种情况由 finders-load-bean 属性控制。

（2）如果将 finders-load-bean 设置为 true，则 finder 方法将返回大量实例。

① 所有返回的实例都将加载到缓存中。

② 这可能就会导致缓存满情况发生。

（3）因此如果使用了 CMP，则在构造和配置实体 bean finder 方法时要小心谨慎。

（4）将 finders-load-bean 设置为 false，这样在调用第一个方法时才会将 bean 加载到缓存中。

如果将 finders-load-bean 设置为 true，则需确保 finder 方法只返回少量实例，而且缓存大到可以容纳所有返回的实例。

3．实体 bean 关联缓存加载

（1）关联缓存加载。

① 将相关的 bean 加载到缓存中以提高性能。

② 发出针对相关 bean 的联合查询以避免进行多次查询。

③ 由 weblogic-cmp-rdbms-jar.xml 文件中的 relationship-caching 标记控制。

④ 如果使用时不够小心，也会导致缓存满情况发生。

（2）请检查：

① 为应用程序设置的适当的关联缓存加载。

② 如果启用了该功能，缓存中允许的最大 bean 数足以容纳加载的关联 bean。

4．有状态会话 bean 缓存类型

（1）可按以下方法管理有状态会话 bean 的缓存。

① 在 weblogic-ejb-jar.xml 中设置 cache-type。

② 紧密钝化或近期不常用（Least Recently Used，LRU）。

在某个 bean 达到其 idle-timeout-seconds 时，即使缓存未满也钝化该 bean。

③ 松散钝化或近期未用（Not Recently Used，NRU）。

a. 是默认值，它会尽可能避免钝化。

b. 如果缓存中的 bean 数接近 max-beans-in-cache，即使 bean 空闲时间尚未过期也钝化 bean。

5．有状态会话 bean

（1）WLS 在下列情况下钝化有状态会话 bean。

① 达到 idle-timeout-seconds 时。

② 达到 max-beans-in-cache 时。

（2）不过，应用程序应该：

① 在不再需要有状态会话 bean 时将它们删除。

② 这样可使 bean 尽快符合钝化条件。

③ 强烈建议采用此方法！

（3）如果应用程序未能有效地删除 bean，则 bean 钝化次数增加就是即将发生缓存满情况的故障症状！

6．内存不足错误

（1）每个有状态会话 bean 或实体 bean 类型均使用单独的缓存，每个缓存都有已定义的最大值。

（2）bean 将继续被加载到缓存中。

① 直至达到 max-beans-in-cache 为止。

② 之后，某些 bean 将被删除或钝化，以为新 bean 让出空间。

③ 因此日常缓存大小实际上就是 max-beans-in-cache!

（3）如果 JVM 堆不足以容纳所有缓存的最大值，将发生 OutOfMemoryError。

（4）内存不足问题的解决办法包括：①减小 max-beans-in-cache 值以适合现有堆大小；②增加 JVM 堆大小以容纳各缓存最大值之和；③ 使用合并的应用程序级别缓存来节约和重叠使用缓存。

（5）要使系统获得最佳效果，可能需要组合使用上述方法！

7．应用程序级别缓存加载

应用程序缓存加载包括以下几方面。

（1）允许同一企业应用程序的多个实体 bean 共享单个运行时的缓存。

（2）能够更有效地利用内存和堆空间。

（3）可以减少缓存数和配置工作量以及简化管理、调整和执行。

（4）不过，由于使用单个线程管理整个应用程序缓存，因此可能不适合吞吐量高的情况。

（5）如果各任务争相取得对线程的控制，可能会导致瓶颈发生。

18.5　故障排除检查清单

（1）精确定位出现故障症状的缓存。

（2）监视缓存，看是否有其他缓存参与。

（3）根据问题是实体 bean 缓存问题还是有状态会话 bean 缓存问题，确定问题的发生原因。

（4）使用缓存配置参数和/或应用程序来解决问题。

（5）继续监视，看是否还会发生问题。

第 19 章　Java 虚拟机 GC 及其相关问题

19.1　JVM 的 GC 概述

GC 即垃圾收集机制是指 JVM 用于释放那些不再使用的对象所占用的内存。Java 语言并不要求 JVM 有 GC，也没有规定 GC 如何工作。不过常用的 JVM 都有 GC，而且大多数 GC 都使用类似的算法管理内存和执行收集操作。

在充分理解了垃圾收集算法和执行过程后，才能有效地优化它的性能。不同的应用程序可能需要采用不同的垃圾收集策略。比如，实时应用程序主要是为了避免垃圾收集导致应用响应中断，而大多数 OLTP 应用程序则更注重整体效率、吞吐量。理解了应用程序的工作负荷和 JVM 支持的垃圾收集算法，便可以进行优化配置垃圾收集器。

垃圾收集的目的在于清除不再使用的对象。GC 通过确定对象是否被活动对象引用来确定是否收集该对象。GC 首先要判断该对象是否是时候可以收集。两种常用的方法是引用计数和对象引用遍历。

19.2　回顾：JVM 的内存管理及 GC 算法

所有的数据和程序都是在运行数据区存放，它包括以下几部分。

19.2.1　栈内存 Stack

栈也叫栈内存，是 Java 程序的运行区，是在线程创建时创建的，它的生命期跟随线程的生命期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收问题，只要线程一结束，该栈就 Over。问题出来了：栈中存的是哪些数据呢？又是什么格式呢？

栈中的数据都是以栈帧（Stack Frame）的格式存在，栈帧是一个内存区块，是一个数据集，是一个有关方法（Method）和运行期数据的数据集，当一个方法 A 被调用时就产生了一个栈帧 F1，并被压入到栈中，A 方法又调用了 B 方法，于是产生栈帧 F2 也被压入栈，执行完毕后，先弹出 F2 栈帧，再弹出 F1 栈帧，遵循“先进后出”原则。

那栈帧中到底存在着什么数据呢？栈帧中主要保存 3 类数据：本地变量（Local Variables），包括输入参数和输出参数以及方法内的变量；栈操作（Operand Stack），记录出栈、入栈的操作；栈帧数据（Frame Data），包括类文件、方法等。光说比较枯燥，我们画个图来理解一下 Java 栈，如图 19-1 所示。

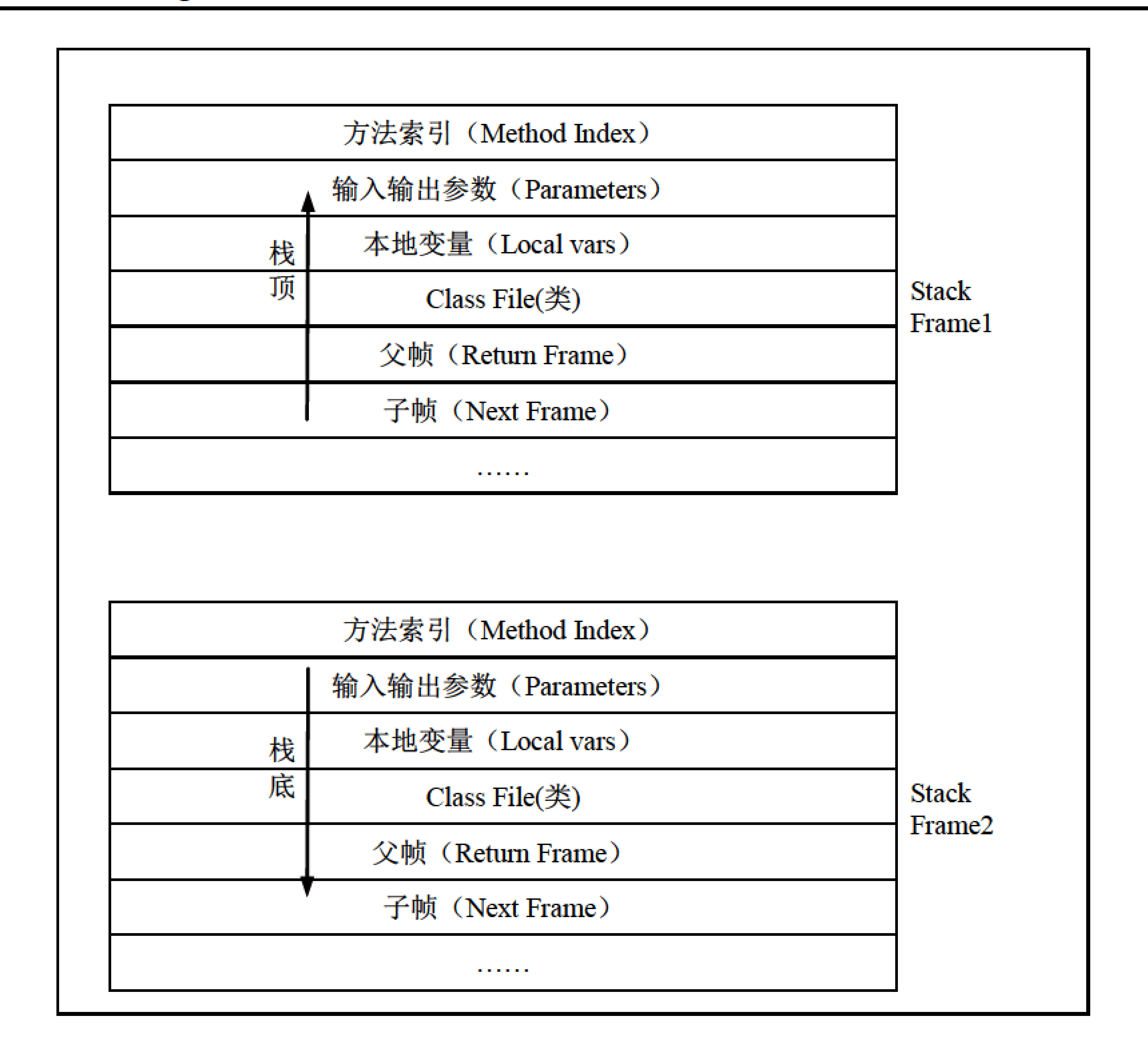


图 19-1

19.2.2　堆内存 Heap

生命周期很短的对象归为 Young Generation，这部分对象在 GC 的时候，很大部分已经成为非活动对象；Young Generation 的 GC 称为 minor GC。经过数次 Minor GC，依旧存活的对象将被移出 Young Generation，移到 Tenured Generation；Young Generation 分为：① Eden，每当对象创建的时候，总是被分配到这个区域；② Survivor 0，from Space 空间；③ Survivor 1，to Space 空间。

注：Survivor 0 和 Survivor 1 总有一个为空，其身份在每次 Minor GC 后被互换，Minor GC 的时候，会把 Eden+Survivor 0(1)的对象 Copy 到 Survivor 1(0)。

1．Permanent Space（永久存储区）

永久存储区是一个常驻内存区域，用于存放 JDK 自身所携带的 Class、Interface 的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据一般是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存。

2．Young Generation Space（新生区）

新生区是类的诞生、成长、消亡的区域，一个类在这里产生、应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分：伊甸区（Eden Space）和幸存者区（Survivor Pace），

所有的类都是在伊甸区被 new 出来的。幸存区有两个： 0 区（Survivor 0 Space）和 1 区（Survivor 1 Space）。当伊甸园的空间用完时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收，将伊甸园区中的不再被其他对象所引用的对象进行销毁。然后将伊甸园中的剩余对象移动到幸存 0 区。若幸存 0 区也满了，再对该区进行垃圾回收，然后移动到 1 区。那如果 1 区也满了呢？再移动到养老区。

3．Tenure Generation Space（养老区）

养老区用于保存从新生区筛选出来的 Java 对象，一般池对象都在这个区域活跃。3 个区的示意图如图 19-2 所示。



图 19-2

19.2.3 常用的 GC 算法

1．引用计数法（Reference Counting Collector）

引用计数法是唯一没有使用根集的垃圾回收的法，该算法使用引用计数器来区分存活对象和不再使用的对象。一般来说，堆中的每个对象对应一个引用计数器。当每一次创建一个对象并赋给一个变量时，引用计数器置为 1。当对象被赋给任意变量时，引用计数器每次加 1。当对象出了作用域后（该对象丢弃不再使用），引用计数器减 1。一旦引用计数器为 0，对象就满足了垃圾收集的条件。

引用计数为每一个对象提供了当前被引用的次数。前者的显式便利了系统中所有的指针得到可用的对象集合，后者便利每一个对象，判断对象是否仍被引用，即对象是否还将被使用。引用计数的一个重要的好处是它的实时性，因为每一个操作都可以及时地修改对象的引用计数。从而在语言的层面上了实现 GC。而且其他的执行进程不需要被挂起。问题是 GC 本身需要占用多余的存储空间。

当程序创建一个循环引用关系的时候，环中的对象的引用计数永远不会降为 0，也就永远无法被回收，这种情况下，引用计数算法将会失效。

效率的问题源于引用计数的本质，它的耗费与处理时间成正比（Per Operation Update）。这个问题对于栈上的生存期较短的对象尤其显著。延迟的引用计数技术可以有效地提高效率，然而其占用的处理时间仍然是成比例的，只是比例相比下降了很多。

对象的回收：引用计数为 0 的对象将会被添加到一个回收链表中，等待回收进程回收。

基于引用计数器的垃圾收集器运行较快，不会长时间中断程序执行，适合必须实时运行的程序。但引用计数器增加了程序执行的开销，因为每次对象赋给新的变量时，计数器加 1，而每次有对象出了作用域时，计数器就减 1。

2. Tracing 算法（Tracing Collector）

Tracing 算法是为了解决引用计数法的问题而提出的，它使用了根集的概念。基于 Tracing 算法的垃圾收集器从根集开始扫描，识别出哪些对象可达，哪些对象不可达，并用某种方式标记可达对象，例如，对每个可达对象设置一个或多个位。在扫描识别过程中，基于 Tracing 算法的垃圾收集也称为标记和清除（Mark-and-Sweep）垃圾收集器。

MS 算法首先通过 Root Set 得到当前的可达集并对相应的对象做出标记（Mark），然后将非可达集添加到回收链表中（Sweep）。

MS 的第一个问题是它容易造成零碎的内存，小对象往往散步在堆中，造成难以创建大对象。（事实上这个问题对于所有的 GC 算法都存在）。

MS 第二个问题也是它的效率问题，其耗费与堆大小（活动对象与垃圾对象的总合）成正比。

MS 的第三个问题是对象局部性（Locality），因为对象不会被移动，造成堆上存在大量空隙。

3. Compacting 算法（Compacting Collector）

为了解决堆碎片问题，基于 Tracing 的垃圾回收吸收了 Compacting 算法的思想，在清除的过程中，算法将所有的对象移到堆的一端，堆的另一端就变成了一个相邻的空闲内存区，收集器会对它移动的所有对象的所有引用进行更新，使得这些引用在新的位置能识别原来的对象。在基于 Compacting 算法的收集器的实现中，一般增加句柄和句柄表。

标记—压缩算法（MC）与 MS 算法的第一步骤相同，在第二步骤中，MC 移动所有的活动对象，使它们在堆上相邻的位置存在，从而形成连续的空闲内存。这个过程类似于 Windows 的内存碎片整理。

MC 的问题显而易见，对于大量临时对象的情况下，其性能远低于 MS 算法。

MC 与 MS 在第一个阶段非常类似，但是不同的是 MC 并不回收垃圾，而 MS 往往进行回收垃圾。

4. Copying 算法（Copying Collector）

该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。它开始时把堆分成一个活动面和多个空闲面，程序从活动面为对象分配空间，当对象满了，基于 Copying 算法的垃圾收集就从根集中扫描活动对象，并将每个活动对象复制到空闲面（使得活动对象所占的内存之间没有空闲洞），这样空闲面就变成了活动面，原来的活动面也就变成了空闲面，程序会在新的活动面中分配内存。

Copying 回收合并了 MC 算法中的扫描与压缩两个过程，它的目标也是将所有的活动对象移动到相邻的空间中，从而形成连续的可用内存。

最常见的 Copying 算法将堆分为两个区域，在运行过程中同时仅有一个区域可以被引用。在每一次整理过程中，使用区域的对象将会使用广度优先的遍历算法（内存中的引用关系是树状的结构）被复制到空闲区域中连续的存储空间中。

Copying 的最大问题是它要求实际需要的近两倍的内存。增加可用的内存可以显著地提高 Copying 算法的效率；当可用内存不足的时候，Copying 的次数将会增加，提高 GC 对系统性能的影响。

一种典型的基于 Copying 算法的垃圾回收是 Stop-and-Copy 算法，在活动面与空闲区域面的切换过程中，程序暂停执行。

缺陷：收集器必须复制所有的活动对象，这增加了程序的等待时间，这是 Copying 算法低效的原因。

5. Non-Copying Implicit Collection（非复制隐式）

非复制隐式的内存收集算法为每一个对象增加了两个指针和一个颜色域，从而将内存中的所有对象组成了一个双向链表的结构（颜色域标志当前对象处于哪一个内存节点-bulk 中）。与 Copying 相比，活动对象并不会在整理过程中被移动，所被改动的仅仅是对象的 3 个附加域。因此对象的回收可以在常数时间内被完成，NCIC 的消耗与系统中的对象的数目成正比。

NCIC 提高了对象遍历的效率，尤其是大对象的便利效率，但是它无法避免稀疏内存的问题。

19.3　GC 统计信息

GC 输出参数格式如下。

-verbose:gc 在虚拟机发生 GC 时在输出设备显示信息。

完整 GC 将产生类似如下内容的消息：

```
[memory ] <start>: GC <before>K-><after>K (<heap>K), <total> ms
```

其中：

<start>，GC 的开始时间（秒），从 JVM 启动开始计算。

<before>，回收前对象所使用的内存（KB）。

<after>，回收后对象所使用的内存（KB）。

<size>，回收后的堆大小（KB）。

<total>，执行回收的总时间（毫秒）。

例如：

```
[memory ] 7.160: GC 131072K->130052K (131072K) in 1057.359 ms
```

显示在执行 GC 后仍存在碎片的 GC 消息示例如下。

示例 19-1：

```
[memory ] 8.162: GC 73043K->72989K (131072K) in 12.938 ms
[memory ] 8.172: GC 72989K->72905K (131072K) in 12.000 ms
[memory ] 8.182: GC 72905K->72580K (131072K) in 13.509 ms
. . .
java.lang.OutOfMemoryError
```

上面示例显示 GC 减少了使用的堆，其大小已与最大堆大小有明显差距，依然出现 OOME！

-Xloggc:<filename> 指定 GC 日志记录输出的文件路径。

e.g. 使用 -Xloggc:filename 旗标的 GC 日志记录如下。

示例 19-2：

```
69.713: [GC 11536K->11044K(11916K), 0.0032621 secs]
69.717: [Full GC 11044K->5143K(12016K), 0.1429698 secs]
69.865: [GC 5958K->5338K(11628K), 0.0021492 secs]
69.872: [GC 6169K->5418K(11628K), 0.0021718 secs]
69.878: [GC 6248K->5588K(11628K), 0.0029761 secs]
69.886: [GC 6404K->5657K(11628K), 0.0017877 secs]
```

在这个例子中，JVM 在这个 GC 周期开始之前运行了 69.713 秒。从左到右的字段为：执行的收集的类型、GC 之前的堆使用、总的堆能力和 GC 事件的持续时间。从这个描述中我们可以看出第一个 GC 事件是一个小的收集。在 GC 开始之前，使用了 11536KB 的堆空间。在完成时，使用了 11044KB，堆能力为 12016KB，而整个收集用了.0032621 秒。下一个事件，一个完全的 GC，在 69.717 秒时或者上一个小 GC 事件之后 0.003 秒时开始。注意，如果将小 GC 事件的持续时间加到其开始时间上，就会看到它在完全的 GC 开始之前不到 1 毫秒结束。因此我们可以得出结论：小收集没有恢复足够的空间，这种失败触发了完全的 GC。对应用程序来说，这像是一个持续了 0.1462319 秒的事件。

-XX:+PrintGC：输出 GC 信息，类似-verbose:gc 参数。输出形式如下。

```
[GC 118250K->113543K(130112K), 0.0094143 secs]
[Full GC 121376K->10414K(130112K), 0.0650971 secs]
```

-XX:+PrintGCDetails，打印输出更多垃圾回收相关信息。

e.g.串行垃圾收集器使用该参数打印出来的信息：

```
[GC [DefNew: 64575K->959K(64576K), 0.0457646 secs] 196016K->133633K(261184K
```

这个信息显示，这次收回了 98% 的 DefNew 年轻代的数据 64575K->959K(64576K) 并在其上消耗了 0.0457646 secs（大约 45 毫秒），整个堆的占用率下降了大约 51% 196016K->133633K(261184K)，而且通过最终的时间 0.0459067 secs 显示在垃圾收集中有轻微的开销（在年轻代之外的时间）。

e.g.下面是一个 concurrent collection 的输出信息（部分省略），可以参考一下。

示例 19-3：

```
[GC [1 CMS-initial-mark: 13991K(20288K)] 14103K(22400K), 0.0023781 secs]
```

```
[GC [DefNew: 2112K->64K(2112K), 0.0837052 secs] 16103K->15476K(22400K),
0.0838519 secs]
...
[GC [DefNew: 2077K->63K(2112K), 0.0126205 secs] 17552K->15855K(22400K),
0.0127482 secs]
[CMS-concurrent-mark: 0.267/0.374 secs]
[GC [DefNew: 2111K->64K(2112K), 0.0190851 secs] 17903K->16154K(22400K),
0.0191903 secs]
[CMS-concurrent-preclean: 0.044/0.064 secs]
[GC[1 CMS-remark: 16090K(20288K)] 17242K(22400K), 0.0210460 secs]
[GC [DefNew: 2112K->63K(2112K), 0.0716116 secs] 18177K->17382K(22400K),
0.0718204 secs]
[GC [DefNew: 2111K->63K(2112K), 0.0830392 secs] 19363K->18757K(22400K),
0.0832943 secs]
...
[GC [DefNew: 2111K->0K(2112K), 0.0035190 secs] 17527K->15479K(22400K),
0.0036052 secs]
[CMS-concurrent-sweep: 0.291/0.662 secs]
[GC [DefNew: 2048K->0K(2112K), 0.0013347 secs] 17527K->15479K(27912K),
0.0014231 secs]
[CMS-concurrent-reset: 0.016/0.016 secs]
[GC [DefNew: 2048K->1K(2112K), 0.0013936 secs] 17527K->15479K(27912K),
0.0014814 secs]
```

-XX:+PrintGCTimeStamps，输出每次开始时间的时间戳，用于查看垃圾回收频率。

示例 19-4：

```
111.042: [GC 111.042: [DefNew: 8128K->8128K(8128K), 0.0000505 secs] 111.042:
[Tenured: 18154K->2311K(24576K), 0.1290354 secs] 26282K->2311K(32704K),
0.1293306 secs]
```

信息显示，垃圾回收在程序运行后 111 秒开始，小回收同时启动。信息中还显示了主回收中的年老代的垃圾回收信息。年老代的空间使用率下降了大约 10% 18154K->2311K(24576K)，用时 0.1290354（大约 130 毫秒）。

19.4　JVM 常用命令行参数设置

19.4.1　通用参数

-Xmx 设置最大堆大小，如果与活动数据量相比，最大堆大小值设置得较低，则会因强制进行频繁垃圾收集而降低性能。

-Xms 设置最小堆大小，建议将最小堆大小（-Xms）与最大堆大小（-Xmx）设置为相同的值，以便将垃圾收集的消耗降至最低。

e.g.从 Java 命令行启动 WebLogic Server 实例时，可使用以下命令指定 BEAJRockitVM 堆大小值：

```
$ java -Xms512m -Xmx512m
```

这些值的默认大小的度量单位是字节。该值后附加字母“k”或“K”表示 KB；附加“m”或“M”表示 MB；附加“g”或“G”表示 GB。上例将 512 MB 内存分配给运行在 JVM 中的 WebLogic Server 实例的最小和最大堆值。

19.4.2 Java 虚拟机几个命令行参数说明

JVM 常用命令如下。

（1）

```
-Xmixed              mixed mode execution (default)
```

混合模式执行。

（2）

```
-Xint                interpreted mode execution only
```

解释模式执行。

（3）

```
-Xbootclasspath:<directories and zip/jar files separated by ;>
      set search path for bootstrap classes and resources
```

设置 zip/jar 资源或者类（.class 文件）存放目录路径。

（4）

```
-Xbootclasspath/a:<directories and zip/jar files separated by ;>
      append to end of bootstrap class path
```

追加 zip/jar 资源或者类（.class 文件）存放目录路径。

（5）

```
-Xbootclasspath/p:<directories and zip/jar files separated by ;>
      prepend in front of bootstrap class path
```

预先加载 zip/jar 资源或者类（.class 文件）存放目录路径。

（6）

```
-Xnoclassgc          disable class garbage collection
```

关闭类垃圾回收功能。

（7）

```
-Xincgc              enable incremental garbage collection
```

开启 Incremental 垃圾回收算法。

（8）

```
-Xloggc:<file>    log GC status to a file with time stamps
```

记录垃圾回日志到一个文件。

（9）

```
-Xbatch           disable background compilation
```

关闭后台编译。

（10）

```
-Xms<size>        set initial Java heap size
```

设置 JVM 初始化堆内存大小。

（11）

```
-Xmx<size>        set maximum Java heap size
```

设置 JVM 最大的堆内存大小。

（12）

```
-Xss<size>        set java thread stack size
```

设置 Java 线程的栈内存大小。

（13）

```
-Xprof            output cpu profiling data
```

输入 CPU profiling 数据。

（14）

```
-Xfuture          enable strictest checks, anticipating future default
```

执行严格的代码检查，预测可能出现的情况。

（15）

```
-Xrs              reduce use of OS signals by Java/VM (see documentation)
```

忽略操作系统信号。

（16）

```
-Xcheck:jni       perform additional checks for JNI functions
```

对 JNI 函数执行检查。

（17）

```
-Xshare:off       do not attempt to use shared class data
```

尽可能不去使用共享类的数据。

（18）

```
-Xshare:auto      use shared class data if possible (default)
```

尽可能的使用共享类的数据。

（19）

```
-Xshare:on        require using shared class data, otherwise fail.
```

尽可能使用共享类的数据，否则运行失败。

19.4.3　Sun 的 JVM 参数

1．Serial Collector 相关参数

-XX:+UseSerialGC 设置 GC 为串行收集器，JDK 5.0 以前默认均为串行收集器。

2．Throughput Colletor 相关参数

-XX:+UseParallelGC 指定在 New Generation 使用 Parallel Collector（并行收集），不能和-XX:+UseConcMarkSweepGC 一起使用。

-XX:+UseParNewGC 指定在 New Generation 使用 Parallel Collector，是 UseParallelGC 的升级版本，有更好的性能，可与-XX:+UseConcMarkSweepGC 一起使用。

-XX:ParallelGCThreads=<desired number> 指定 Parallel Collection 时启动的 thread 个数，默认是物理 processor 的个数。

-XX:MaxGCPauseMillis=<nnn>设置最大暂停时间。

-XX:GCTimeRatio=<nnn>设置 GC 时间与应用程序时间之间的比率为 1/(1+<nnn>)。

e.g. : -XX:GCTimeRatio=19 则 GC 时间占总时间的 5%。

-XX:YoungGenerationSizeIncrement=<Y>设置年轻代增量比例。

-XX:TenuredGenerationSizeIncrement=<T>设置年老代增量比例。

-XX: AdaptiveSizeDecrementScaleFactor=<D>设置缩小比例。

注：如果增量是 X%，那么每次减少量就是（X/D）%。

3．The Concurrent Low Pause Collector 相关参数

-XX:+UseConcMarkSweepGC 指定在 Old Generation 使用 The Concurrent Low Pause collector。

-XX:CMSInitiatingOccupancyFraction=<nn> 指示在 Old Generation 使用了 n%的比例后，启动 Concurrent Collector，默认值是 68。

-XX:+CMSIncrementalMode 设置为增量模式，默认为 Disabled。

-XX:+CMSIncrementalPacing 启动自动调整增量模式参数，默认为 Disabled。

-XX:CMSIncrementalDutyCycle=<N>设置允许 Minor Collection 运行时间所占百分比，默认为 50。

-XX:CMSIncrementalDutyCycleMin=<N>（用于 CMSIncrementalPacing 被设为 Enabled 的情况下）设置 Minor Collection 运行时间百分比的最低底线，默认为 10。

-XX:CMSIncrementalSafetyFactor=<N>用来计算循环的次数，默认值为 10。

-XX:CMSIncrementalOffset=<N> 在小回收之间，增量模式中占空比开始的时间，或者说是向右的平移量，默认值为 0。

-XX:CMSExpAvgFactor=<N>当进行并发回收统计，计算指数平均值时，当前采样所用的权值，默认为 25。

-XX:+UseCMSCompactAtFullCollection 开启 CMS 阶段进行合并碎片功能，因为 CMS 不会整理堆碎片。

-XX：ParallelCMSThreads=20 设置 CMS 启动的回收线程数目，默认为（ParallelGCThreads+3）/4。

-XX:+CMSScavengeBeforeRemark 强制 Remark 之前进行一次 Minor GC，减少 Remark 的暂停时间。

4．Incremental Collector 相关参数

-XX:UseTrainGC 设置 GC 为 Incremental Collector，不能与-XX:+UseParallelGC 和 -XX:+UseParNewGC 一起使用，这个收集器已经不再发展，将来的版本将不再支持它。

5．堆大小设置

-XX:NewRation=n 设置 Tenured Generation 与 Young Generation 所占堆大小比率，如 n=3,则 tenured to young ratio 为 1：3。

-XX:MinHeapFreeRatio=<minimum> 最小堆内存空闲百分比，默认为 40，即当空闲堆内存所占堆内存百分比低于 40%，JVM 就会试图扩张堆内存空间。

-XX:MaxHeapFreeRation=<maximum> 最大内存空闲百分比，默认为 70，即当空闲堆内存所占堆内存百分比高于 70%，JVM 就会试图压缩堆内存空间。

-XXSurvivorRation=n 设置新堆大小比率，年轻代中 Eden 区与两个 Survivor 区的比值。例如：n 为 3，表示 Eden：Survivor 等于 3：2，一个 Survivor 区占整个年轻代的 1/5。

-XX::GCTimeRatio=<nnn> 设置垃圾回收时间与非垃圾回收时间的比值，即吞吐量，公式为 1、（1+N）例如：-XX:GCTimeRatio=19 时，表示 5%的时间用于垃圾回收。

-XX:Newsize=n 设置新一代堆大小的下限，将 -XX:NewSize 的大小设置为堆大小的四分之一，这是一条通用规则。如果存在大量短期对象，则增大此选项的值。增加 Java 新对象生产堆内存相当于增加了处理器的数目。并且可以并行地分配内存，但是请注意内存的垃圾回收却是不可以并行处理的。

-XX:MaxNewSize 设置新一代堆大小的上限。

-XX:MaxPermSize 设置持久代大小，是经常的 JVM 调优参数。

注：-XX:MaxPermSize 不适用于 IBM 的 JDK，在 Sun、HP、Solaris、Linux 上均支持此参数。

6．Full Collection 设置

-XX:+DisableExplicitGC 禁止 Java 程序触发的 Full GC 操作，如 System.gc()的调用，

防止在程序里误用了，对系统性能造成影响。

-XX:SoftRefLRUPolicyMSPerMB 指定每兆字节堆空闲空间的 Soft Reference 保持存活（一旦它不强可达了）的毫秒数，默认为 1000。

19.4.4 IBM 的 JVM 参数

-verbose:gc 输出 GC 过程中的信息，例如下边是运用-verbose：gc 对应用中调用 System.gc()的信息输出。

示例 19-5：

```
<GC(3): GC cycle started Tue Mar 19 08:24:34 2002
<GC(3): freed 58808 bytes, 27% free (1163016/4192768), in 14 ms>
<GC(3): mark: 13 ms, sweep: 1 ms, compact: 0 ms>
<GC(3): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
```

GC(3)表示在 JVM 中是第三轮 GC，第一行展示了启动的时间，是 2002 年 3 月 19 日 08:24:34；第二行显示在 14ms 内释放了 58808 字节，占空间栈堆的 27%，1163016 是现有空闲空间字节数，4192768 是堆中一共的字节数；第三行分别显示了记录、回收、压缩阶段所用的时间，由于没有进行压缩，所以 compact 值为 0，最后一行指在 GC 过程中发现的对象引用，在这个例子中没有对象被找到。

-Xcompactgc 开启在每轮 GC 过程中进行压缩，默认值为 false。

-Xgcpolicy:<optthruput | optavgpause> 默认将 gcpolicy 的值赋给 optthruput。将 gcpolicy 值赋给 optthruput，不能进行并行标记，如果在不同应用程序下没有暂停时间的困扰，则可以设置 optthruput，以获得最好的吞吐量；将 gcpolicy 赋给 optavgpause，能够进行并发标记，如果应用程序有 GC 引起的时间响应问题，可以赋值给 optavgpause 减轻这个问题，但是会消耗一些吞吐量。

-Xgcthreads<*number*> 设置总共的线程的数量，在多进程系统中，当重置模式时，该参数默认值为 1，当不重置模式时，默认值为 n。

-Xinitacsh<size> 设置 application-class system 堆的初始大小，这个参数只在重置 JVM 时有用。在这个堆中的类在 JVM 存在该参数的生命时间，在 ResetJavaVM()过程中重置，在应用程序中可以连续使用。默认值为 128KB（32 位机），8MB（64 位机）。例如，-Xinitacsh256k 初置堆大小为 256KB。

-Xinitsh<*size*> 设置系统堆的初始大小。这个系统堆不受 GC 的影响，最大值可以设置到非常大，默认为 128KB（32 位机型），8MB（64 位机型）。例如，-Xinitsh256k 初置系统堆大小为 256KB。

19.5 JVM 性能优化

通常情况下是不建议在没有任何统计和分析的情况下去手动配置 JVM 的参数来调整

性能，因为在 JVM 5 以上已经做了根据机器和 OS 的情况自动配置合适参数的算法，基本能够满足大部分的情况，当然这种自动适配只是一种通用的方式，如果说真的要达到最优，那么还是需要根据实际的使用情况来手动地配置各种参数，提高性能。JVM GC 调整优化是一个极为复杂的过程，由于各个程序具备不同的特点，如 Web 和 GUI 程序就有很大区别（Web 可以适当的停顿，但 GUI 停顿是客户无法接受的），而且由于所运行的机器的配置不同（主要 CPU 个数/主频、内存不同），所以使用的 JVM GC 种类也会不同。

19.5.1　优化目标

（1）减少 Minor GC 次数。
（2）缩短 Minor GC 时间。
（3）减少 Full GC 的次数，尽量使 Full GC 不发生。

19.5.2　如何设置 GC

（1）大部分情况不用调用 GC，让 JVM 自己做好了。
（2）任何时候都是让 JVM 先自己选择 GC，当性能有问题的时候再手工调。
（3）当有 Pauses 时间要求的时候，尝试 Concurrent GC。
（4）当没有 Pauses 时间要求时，尝试 Parallel Collector。
（5）合理设置 GC 其他参数以及堆的其他参数。
（6）不要迷信任何主观的想法，一定要测试、比较、修改、测试，反复做下去，直到性能在合理的期望中。
（7）Server Mode 时内存的划分反向了，用 jsonsole 可以观察到。

19.5.3　如何监视 JVM GC

在 Java 程序启动的 opt 里加上如下几个参数。

示例 19-6：

```
-XX:-PrintGC   在日志中记录 GC 事件的信息
-XX:-PrintGCDetails 在日志中记录详细的 GC 信息
-XX:-PrintGCTimeStamps 记录 GC 日志的时间
当把-XX:-PrintGCDetails 加入到 javaopt 里以后可以看见如下输出：
[GC[DefNew:34538K->2311K(36352K),0.0232439secs]45898K-
>15874K(520320K),0.0233874secs]
[FullGC[Tenured:13563K->15402K(483968K),0.2368177secs]21163K-
>15402K(520320K),[Perm:28671K->28635K(28672K)],0.2371537secs]
```

它们分别显示了 JVM GC 的过程，清理出了多少空间。第一行 GC 使用的是普通 GC（Minor Collections），第二行使用的是 Full GC（Major Collections）。它们的区别很大，在第一行最后我们可以看见它的时间是 0.0233874 秒，而第二行的 Full GC 的时间是 0.2371537

秒。第二行的时间是第一行的接近 10 倍，也就是我们这次调优的重点，减少 Full GC 的次数，以为 Full GC 会暂停程序比较长的时间，如果 Full GC 的次数比较多。程序就会经常性地假死。

19.5.4 性能优化

可以通过上述 JVM 参数设置适合自己应用程序的 GC，如下面几种办法。

1. 对于吞吐量的调优

4GB 的内存，32 个线程并发能力， 64 位操作系统。

```
java -Xmx3800m -Xms3800m -Xmn2000 -Xss128k -XX:+UseParallelGC -XX:
ParallelGCThreads=20
```

-Xmx3800m -Xms3800m 配置了最大 Java Heap 来充分利用系统内存。

-Xmn2000 创建足够大的年轻代（可以并行被回收）充分利用系统内存，防止将短期对象复制到老年代。

-Xss128 减少默认最大的线程栈大小，提供更多的处理虚拟内存地址空间被进程使用。

-XX:+UseParallelGC 采用并行垃圾收集器对年轻代的内存进行收集，提高效率。

-XX:ParallelGCThreads=20 减少垃圾收集线程，默认是和服务器可支持的线程最大并发数相同，但往往不需要配置到最大值。

2. 尝试采用对年老代并行收集

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:
ParallelGCThreads=20 -XX:+UseParallelOldGC
```

-Xmx3550m -Xms3550m 内存分配被减小，因为 ParallelOldGC 会增加对于 Native Heap 的需求，因此需要减小 Java Heap 来满足需求。

-XX:+UseParallelOldGC 采用对于年老代并发收集的策略，可以提高收集效率。

3. 提高吞吐量，减少应用停顿时间

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+
UseConcMarkSweepGC -XX:+UseParNewGC -XX:SurvivorRatio=8 -XX:TargetSurv-
ivorRatio=90 -XX:MaxTenuringThreshold=31
```

-XX:+UseConcMarkSweepGC -XX:+UseParNewGC 选择了并发标记交换收集器，它可以并发执行收集操作，降低应用停止时间，同时它也是并行处理模式，可以有效地利用多处理器的系统的多进程处理。

-XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=31 表示在年轻代中 Eden 和 Survivor 的比例，设置增加了 Survivor 的大小，越大的 Survivor 空间可以允许短期对象尽量在年轻代消亡。

-XX:TargetSurvivorRatio=90 允许 90%的空间被占用，超过默认的 50%，提高对于 Survivor 的使用率。

性能优化不需要经常做，除非觉得系统存在相当严重的性能问题；因为优化通常意味着要改动，在系统没有足够多的 test case 和 auto test 保证下，很难保证改动没有负面的影响。现在的硬件已经相当廉价，如果可以，选择增加适当的硬件投入是一个不错的选择。

系统是否存在严重的性能问题，现在的性能测试工具还是蛮多的，比较好用的包括 Load Runner 和 Jmeter。前者可以自动录制测试脚本，后者通过第三方工具也可以录制自动测试脚本。通常测试工具都有格式良好的 report 输出，在 Web 应用中有几个指标需要我们给予更多关注：① Average Hits per Second；② 90 Percent transaction response time；③ Average Throughput（bytes/second）。需要注意的是，指标①和指标②是有一些相互约束关系的，很难做到系统有很高的并发请求量，同时请求的响应时间又很小，就好像很难让“马儿跑得快”又让“马儿不吃草”，我们需要在二者之间找到平衡点。一个性能测试报告，在一定的前提下，比如怎样的硬件设置，怎样的网络环境，怎样的请求数据，怎样的并发模拟，以及多少的并发量，等等，不同的测试环境所得到的数据可能相差甚远。

如果系统确实存在性能问题，比如经常死机，或者响应时间很慢，处理的并发量很小，就不得不进入性能优化阶段。性能优化包括几个环节：运行环境参数调优、数据库调优、应用调优等。

如前边所述，JVM 中提供了丰富的参数供我们设置，合理的设置这些参数，可以有效地提高系统的性能。在 Java 应用中，虽然不太容易出现内存泄漏的问题，因为 JVM 会不定期地进行 GC。但是因为程序的不合理写法，也会导致一些数据不能被收集。典型的状况是在 hashmap 中放置大量不用的数据，而没有及时地清理。在 Web 应用中，很多人喜欢在 session 中放置状态数据，而没有清理，也是内存泄漏的一个原因。在 session 中存放数据还好，因为 session 终究会有过期时间，但是如果在 class 的 static 变量中放置数据，那就没办法了。诊断应用中是否存在内存泄漏也有一些方法，通过分析 JVM GC Log 就是一个直观的方式。通过分析 GC after Heap 的变化趋势，如果 GC after Heap 稳步上升，及时 Full（Major）GC 后，仍然不能降下来，通常就意味着存在内存泄漏了。当然也有情况是，的确有一些数据是 application scope 的，但是要确认除了这些数据，是否还存在一些 unexpected 数据一直占据内存。可以通过 Jprofile 的 memory views 来观察 Class 的对象数，在一段请求过后，如果还存在一些 Class 的 instance 数目相当多，就可以判断这个 Class 可能会是问题的根源。

在 O/R Mapping 盛行的今天，应用代码直接操作数据库已经和我们渐行渐远，我们不用手工写 SQL 查询，一切都是对象操作，除非存在性能问题，否则没有人会乐意关注一些数据库细节。但是不可否认的是，今天的 Java 应用，大多数依然是数据集中式的，需要和数据库频繁地交互，而且数据库也很容易成为性能的瓶颈。数据库的调优包括 3 部分：数据库参数的设置，表结构以及 SQL 代码优化。大多数数据库的参数都相当多，调优所需要的知识很复杂，这也是现在专业的 DBA 特别贵的原因。

至于表结构，因为现在 O/R Mapping 工具已经包揽了这项工作，所以已经转为对象结构的优化了。一般可以适当地非规范化对象结构，包括允许一些冗余属性，同时减少一些双向关联，过往的经验表明，双向关联通常都是性能的杀手。有一块需要做的是索引，合

理的索引可以成倍地提高数据库性能。不过这一块做起来通常要结合应用中使用的 SQL 进行考虑。索引是一把双刃剑，很多时候可以提高查询的速度，但同样也存在维护成本，导致 update 性能下降。而且是否用得上索引，还要看查询返回的结果集，如果查询的结果集很多，此时使用索引，性能反而可能会下降，因为需要进行两次访问，还不如直接 table scan 来的快。

在 SQL 调优时，可以先挑出性能很差同时又执行频繁的 SQL，可以通过 Jprofile 的 CPU Views 过滤出 JDBC calls。这些 SQL 可以用相应的数据库工具中调优，数据库引擎通常会根据数据库的一些 metadata 和 statistics 生成一个 execution plan，通过这个 execution plan，可以找到优化的途径。通常的思路是，表访问策略（index scan、table scan），表连接策略（sort merge join、nested loop join、hash join），表顺序（驱动表的选择），是否充分并行，等等。通过调整 SQL 的写法会使 DBMS 使用不同的执行计划。当然要得到有效的执行计划，需要及时更新数据库的统计信息（Statistics）；有时候发现索引没用上，后来发现原来是这个原因。此外，SQL 语句还有一些其他好的习惯，简单列出一些。

避免 where 子句中出现 or，事实证明 or 的性能通常不如 in。

避免 like 子句，尤其是通配符%在前面。

避免 where 子句中在 indexed column 中加函数，尽量将函数转移到比较运算符右边去，否则没办法使用到索引。

where 中 column 出现的顺序和 index 的顺序一致。

避免使用 select *，不要偷懒，还是列出自己需要的那些列，即使是获取所有的列。

一句性能优良的 SQL 是不太容易写出的，其实这也是大量 O/R Mapping 工具出现的一个原因，想让 O/R Mapping 生成一个很好的 SQL 不容易，不过想让其生成一个很烂的 SQL 也同样是不容易的，毕竟综合了很多人的经验在其中。

应用的调优需要具体情况具体分析，通常 code review 是一个很好的时机，分析具体代码是否存在性能缺陷。此外，通过 Jprofile 观察，其 CPU Views 列出了所有 method 的执行时间，可以找到其中的性能缺陷，重点分析。这种调整通常涉及到算法的优化和结构的调整。

第 20 章　JMS 消息重发故障

20.1　JMS 简介

JMS 即 Java 消息服务（Java Message Service）应用程序接口，是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间或分布式系统中发送消息，进行异步通信。Java 消息服务是一个与具体平台无关的 API，绝大多数 MOM 提供商都对 JMS 提供支持。

20.2　问题描述

JMS 消息被多次重新发送给接收器、订阅用户。一旦有消息到达 JMS 目标，JMS 服务器试图将其发送给有效用户，然后等待用户确认。从 JMS 服务器的角度来看，在 JMS 消息被确认收到以前，该消息就不被视为“已发送”。无论什么原因，只要 JMS 服务器没有收到消息确认，它就会重新发送消息。

JMS 重新发送故障症状包括以下几方面。

（1）JMS 接收器的 onMessage() 方法被多次执行。

（2）消息接受了多次处理。

（3）服务器的运行速度可能会因多次尝试发送一条或多条“恶性”消息而下降，此类消息：① 是接收器拒绝接收的消息；② 通常因实际消息存在问题而遭到拒绝。

（4）消息可能因暂时性的资源短缺或资源无法使用而遭到拒绝。

20.3　问题定位

20.3.1　为什么 JMS 消息会被重新发送

如果 JMS 服务器认为消息未能成功发送，就会发生 JMS 重新发送，发送的 JMS 消息只有得到确认后，才会将该消息视为已发送。图 20-1 所示的消息 3 已发送，对方收到并返回确认信息，则此时 JMS 服务器就认为消息 3 已发送，而消息 1、2、5、6 均未发送。

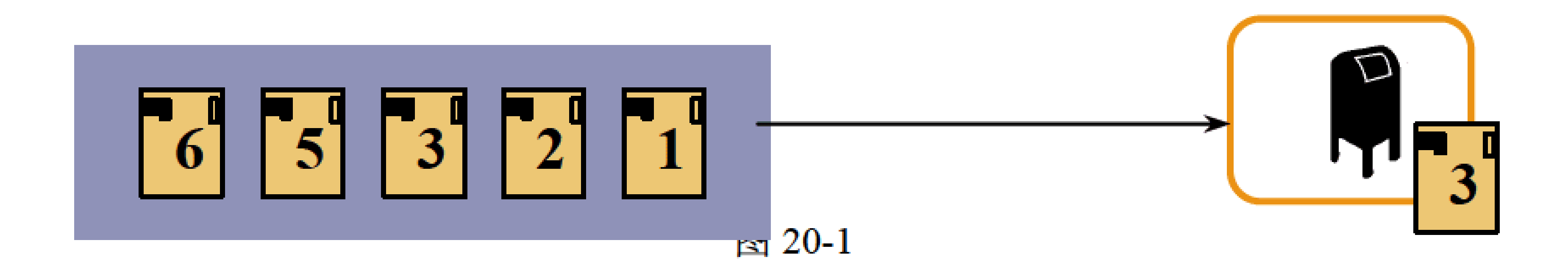


图 20-1

JMS Server 可能会因下列任一原因而重新发送消息。

（1）接收器或 MDB 的 onMessage 方法抛出 Java 异常。例如：

```
java.lang.Error
java.lang.RuntimeException
```

（2）出现“恶性”消息，即这样一些消息。

① 格式不正确并因此遭到拒绝的消息。

② 因暂时性资源中断而遭到拒绝的消息。

（3）客户端接收器（显式 Acknowledge）未调用 session.acknowledgement()。

（4）因出现下列情况而使事务被隐式或显式回滚。

① 参与事务的一个 MDB 因某种原因（例如，下游数据库错误）而失败。

② MDB 是事务性的，但因处理消息时花费的时间过长而引发了超时。

③ 调用了 MDB 的 onMessage()方法中的 ejbcontext.setRollbackOnly()（仅限 CMT）。

④ 独立接收器调用了 session.recover()。

注：JMS 消息重新发送问题大多是由应用程序编码错误引起的！

20.3.2 JMS 重新发送故障的两种类型

（1）在不应重新发送消息时重新发送了消息。

（2）在应该重新发送消息时未重新发送消息。

20.3.3 JMS 重新发送模式问题

（1）JMS 重新发送模式与下列情况下出现的问题有关。

① JMS 服务器认为消息的第一次发送不成功，JMS 服务器随后再次发送该消息。

② JMS 服务器认为第一次发送成功，于是不再重新发送。

（2）发送的 JMS 消息在接受以下操作后视为已发送。

① 确认。

② 在提交了周围事务的情况下，消息可能会得到确认。

20.4 JMS 确认

JMS 消息传送给用户后，JMS 服务器会等待用户的确认，如果 JMS 服务器未收到确

认，会尝试重新发送消息。确认的途径在下面几节内容中展开论述。

20.5　事务会话

20.5.1　使用 JMS 事务会话的操作

（1）将在内部启动一个本地事务，范围限制在会话内的发送、接收操作。

（2）如果接收器对会话进行回滚调用，JMS 服务器将重新发送消息。

20.5.2　JMS 事务会话的适用范围限制

（1）不能参与外部事务，如 bean 管理的或容器管理的（JTA）事务。

（2）在会话范围外不起作用。

20.5.3　容器管理的事务

（1）CMT 的工作方式如下。

① 容器开始事务。

② 调用 onMessage()方法，将消息传递给 MDB。

③ 如果 onMessage()方法成功返回，容器将提交该事务。

④ 如果 onMessage()方法返回异常，容器将回滚该事务。

注：CMT（Container-Managed Transaction，容器管理事务）。

（2）如果事务被回滚，JMS 服务器会立即将消息重新发送给下一位有效用户。

20.5.4　bean 管理的事务

1. BMT

（1）依赖 MDB 应用程序代码来启动、提交或回滚与消息处理有关的事务。

（2）也称做用户事务。

（3）适用于 MDB 用户和独立用户。

注：① BMT（Bean－Managed Transaction，Bean 管理事务）。

② MDB（Message Driven Bean，消息驱动 EJB）。

2. 在 MDB 内部启动事务时

（1）在目标（队列或主题）获取消息不是事务的组成部分。

（2）事务结果对重新发送没有影响，即使事务被回滚也是如此。

20.6 设置确认模式

（1）在创建 JMS 会话时设置。例如：

```
QueueSession qs=qconn.CreatQueueSession(false,Session.Auto_ACKNOWLWDGE)
```

（2）确认模式的选项。

① AUTO_ACKNOWLEDGE，当客户成功地从 receive 方法返回的时候，或者从 MessageListener.onMessage 方法成功返回的时候，会话自动确认客户收到的消息。

② DUPS_OK_ACKNOWLEDGE，该选择只是会话迟钝确认消息的提交。如果 JMS Provider 失败，那么可能会导致一些重复的消息。如果是重复的消息，那么 JMS Provider 必须把消息头的 JMSRedelivered 字段设置为 true。

③ CLIENT_ACKNOWLEDGE 接收器必须显式调用 message.acknowledge()才能确认消息。在这种模式中，确认是在会话层上进行。确认一个被消费的消息将自动确认所有已被会话消费的消息。

例如，如果一个消息消费者消费了 10 个消息，然后确认第 5 个消息，那么所有 10 个消息都被确认。

④ NO_ACKNOWLEDGE 不确认消息。

（3）对于独立的异步监听器，如果消息未得到确认，则只会将消息重新发送一次；如果重新发送失败，则消息将从 JMS Server 中隐式删除。

20.7 诊断 JMS 重新发送问题

20.7.1 应用程序设计

使用程序设计文档或其他工件确定应用程序应如何处理消息，例如：① 接收器是 MDB 还是独立的同步或异步接收器；② 使用何种类型的事务处理，如 CMT、BMT 或者事务会话；③ 应进行何种类型的确认；④ 在什么条件下执行提交、回滚或确认；⑤ 确定设置了哪些处理超时，以及是否在任何情况下均可实现。

20.7.2 应用程序代码诊断

（1）记录接收器 onMessage()方法收到的每条消息。

① 消息 ID。

② 该消息是否被重新发送。

③ 如果消息数量很大，则只记录重新发送的消息。

e.g. 在 onMessage() 方法中实现发送记录的示例代码如下。

示例 20-1：

```
public void onMessage(javax.jms.Messsage msg)
{
  try{
    System.out.println("Msg ID:" + msg.getJMSMessageID());
    System.out.println("Is Message redelivered:"
                              + msg.getJMSRedelivered());
    ...
  }
}
```

（2）捕捉堆栈跟踪和再次抛出导致 onMessage() 方法不能成功返回的异常。

捕捉异常的示例代码如下。

示例 20-2：

```
public void onMessage(Message msg) {
    try {
      TextMessage tm = (TextMessage) msg;
      String text = tm.getText();
      System.out.println("Msg: " + text + " Msg ID:" + msg.getJMSMessageID());
      . . .
    }
    catch(JMSException ex) {
      ex.printStackTrace();
    }
    catch(java.lang.RuntimeException ey) {
      ey.printStackTrace();
          throw ey;
    }
    catch(java.lang.Error ez) {
         ez.printStackTrace();
          throw ez;
    }
}
```

（3）记录下列各项的其他重要事件。

① BMT 和事务会话。

② 事务在事务会话中开始、提交和回滚。

③ 确认模式。

④ 确认和 NACK 将在该模式下被调用。

⑤ CMT，退出 onMessage()方法时记录事务的状态。

⑥ 对事物有影响的调用。

⑦ 如对 ejbcontext.setRollbackOnly() 或 session.recover()的调用。

20.7.3 JMS 调试

（1）使用下列 JMS 调试标志收集有关 JMS 操作所执行的更多信息。

① DebugMessagePath 可帮助确定消息是否被重新发送。

比如，同一消息被连续发送给接收器。

示例 20-3：

```
<Feb 3, 2004 5:01:24 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
BACKEND/BEQueue: Assigning to the backend consumer, message
ID:P<802808.1075856485894.0>>
<Feb 3, 2004 5:01:24 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
BACKEND/BEQueue: Adding backend session's unacked message list, message
ID:P<802808.1075856485894.0>>
<Feb 3, 2004 5:01:24 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
BACKEND/BEQueue: Dispatching to the frontend, message ID:P<802808.1075856485894.0>>
<Feb 3, 2004 5:01:24 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
FRONTEND/FESession (id: <5761803531830905 50.26>) :
Pushing to the client, message ID:P<802808.1075856485894.0>>
>>Print From the onMessage method: I am the MESSAGE, MsgID:
ID:P<802808.1075856485894.0>
…
<Feb 3, 2004 5:02:05 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
BACKEND/BEQueue: Assigning to the backend consumer, message
ID:P<802808.1075856485894.0>>
<Feb 3, 2004 5:02:05 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
BACKEND/BEQueue: Adding backend session's unacked message list,
message ID:P<802808.1075856485894.0>>
<Feb 3, 2004 5:02:05 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
BACKEND/BEQueue: Dispatching to the frontend, message
ID:P<802808.1075856485894.0>>
<Feb 3, 2004 5:02:05 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging
MSG_PATH!
FRONTEND/FESession (id: <5761803531830905 50.34>) :
Pushing to the client, message ID:P<802808.1075856485894.0>>
>>Print From the onMessage method: I am the MESSAGE, MsgID:
ID:P<802808.1075856485894.0>
```

② DebugJMSXA 将有助于确定消息都是因事务相关问题而被重新发送的。

通过 DebugJMSXA 输出示例如下。

示例 20-4：

```
<Feb 3, 2004 5:16:10 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging XA !
XA(24420973,1007511,0000013BC2597F28EDB9) >RM-rollback() >

<Feb 3, 2004 5:16:10 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging XA !
XA(24420973,1007511,0000013BC2597F28EDB9) >TE-recv-startRollback()
(TE-recv hash=31838205 xid=0000013BC2597F28EDB9 mId=<712571.1075857330638.
0> queue=TestDest)>

<Feb 3, 2004 5:16:10 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging XA !
XA(24420973,1007511,0000013BC2597F28EDB9) <TE-recv-startRollback()
(TE-recv hash=31838205 xid=0000013BC2597F28EDB9 mId=<712571.1075857330638.
0> queue=TestDest)OK>

<Feb 3, 2004 5:16:10 PM PST> <Debug> <JMS> <BEA-040002> <JMS Debugging XA !
XA(24420973,26210567,0000013BC2597F28EDB9) >RM-rollback() >
```

（2）应用上述标志的方法：

① 将它们添加到 config.xml 文件的 ServerDebug 部分；

② 在服务器启动脚本中使用-D 标志。

20.8　检查“恶性”消息

（1）检查用于避免“恶性”消息的 JMS 目标参数如下。

① Redelivery Delay Time，指在尝试重新发送消息前消息被搁置的时间。

② Redelivery Limit，指最多可以尝试重新发送消息的次数，尝试次数用尽后，消息将被移至错误目标。

③ Error Destination，指已达到其重新发送极限的消息的目的地（队列或主题）。

（2）检查可能会引发“恶性”消息的条件，如：① 数据库或其他后端资源变为不可用时发生的情况；② 是否在任何情况下都会出现格式不正确的消息。

20.9　故障排除检查清单

（1）与应用程序开发团队协作，以确保：

① 正确处理异常；

② 正确处理确认和/或事物；

③ 安排足够的记录操作来捕捉不正确的处理。

（2）由于重新发送问题大多由应用程序编码错误导致，所以应注意以下两点：

① 先确保应用程序工作正常；

② 如果确信应用程序工作正常，次要可疑点可能就是 JMS 实现本身。

第 21 章　常规 JDBC 问题故障

21.1　JDBC 概述

21.1.1　什么是 JDBC 及其作用

JDBC 是一个标准的 Java API，用于在一个统一的平台中以与供应商无关的方式访问数据库。Java 应用程序可以使用 JDBC 作为驱动访问数据库，执行 SQL 语句。通俗来讲，JDBC 就是 Java 应用程序访问数据库的驱动。

21.1.2　JDBC 驱动程序实现分类

有 4 种主要 JDBC 驱动程序实现，具体如下。

1. 通过客户端 ODBC（Open Database Connectivity）桥接器访问 JDBC 数据库

这种类型的驱动是把所有 JDBC 的调用传递给 ODBC，再由 ODBC 调用本地数据库的 ODBC 驱动来访问数据库。本地数据库的驱动是由数据库厂商提供的，可以用来操作数据库的二进制代码文件。ODBC 是微软公司开放服务结构中有关数据库的一个组成部分，它建立了一组规范，并提供了一组对数据库访问的标准 API，用户可以直接将 SQL 语句发送给 ODBC。

2. 将 JDBC 调用转换为客户端本地数据库调用（本地 API）

这种类型的驱动是把 JDBC 的数据库请求转变为数据库的标准调用，然后调用本地的数据库驱动来访问数据库。

3. 使用网络服务器（例如 WLS）作为 JDBC 请求的代理

这类驱动是把 JDBC 的请求通过网络传递给中间件服务器，中间件服务器再把请求翻译成符合规范的数据库调用，然后再把这种调用传递给数据库服务器。如果中间件服务器也是用 Java 开发的，那么在中间层也可以使用 1、2 型 JDBC 驱动程序作为访问数据库的方法。

4. 直接在客户端部署本地 JDBC

这类驱动是直接把 JDBC 的调用翻译成符合规范的数据库调用，然后发送给数据库服

务器。

第一类驱动，对数据库的调用要先传递给 ODBC，由 ODBC 调用本地数据库的驱动来访问数据库，所以执行效率比较低，而且还必须保证所有的客户段都要安装 ODBC 的驱动和数据库驱动。

跟第一类驱动比，第二类驱动因为不需要经过 ODBC，所以效率有所提高，但是仍然不如第三类和第四类驱动高，而且还需要在客户端安装数据库的客户端。

第三类驱动是基于服务器的，易于维护和升级，而且不需要在客户端安装数据库驱动，不过在服务端仍然需要安装数据库的驱动。

第四类驱动效率最高，而且不需要在客户端或者服务端安装数据库客户端，但是必须根据数据库的不同安装不同的 JDBC 驱动。

21.1.3　WebLogic 常用的 JDBC 驱动

1. 2 类驱动（需要本地库）

包括 WebLogic jDriver for Oracle、WebLogic jDriver for Oracle XA。

第三方驱动，比如 Oracle OCI 和 IBM DB2 驱动。使用场合：用于在本地和分布式事务中连接 WLS 和 DBMS。

2. 4 类驱动（纯 Java）

包括 WebLogic jDrivers for Microsoft SQL Server。

第三方驱动，包括 Oracle Thin 和 Oracle Thin XA 驱动。

使用场合：用于在本地和分布式事务中连接 WLS 和 DBMS。

注：WebLogic jDrivers for Microsoft SQL Server 驱动仅支持本地事务。

3. 3 类驱动

包括 WebLogic RMI 驱动。

使用场合：用于连接外部客户端和 WebLogic Server（Connection Pool）。

21.2　WebLogic 中的 JDBC 配置

WebLogic 的 JDBC 配置中主要包括连接池（Connection Pool）和数据源（Data Source/Multi Data Sources）两部分。到了 9 版本后，将连接池的配置包含进了 Data Source 的配置中。

21.2.1　连接池

1. 什么是连接池

连接池里存放的都是已经建立好的、到特定数据库的物理连接，当应用程序请求访问

数据库时，可以直接使用这些连接访问数据库。

应用程序向连接池中请求数据库连接的时候，WebLogic 服务会先检查有没有空闲的连接，如果有，就把一个空闲连接给应用程序使用，如果没有空闲连接，那就看现在的连接数是否达到了连接池的最大连接数，如果没有达到，则根据设置的容量增长的参数值，增加缓冲池中的数据库连接数，连接池中的数据库连接数已经达到最大值，就会让该请求排队等待，有其他请求释放连接的时候，会让等待队列中的请求使用新释放的连接。如果该请求在队列中的等待时间超过了设置的时间，则该请求退出 JDBC 连接的等待队列。

2．连接池的好处

连接池的好处主要是提高性能并降低系统压力高峰时段对数据库的压力。

（1）在连接池启动时，会创建指定数量的到数据库的物理连接，当应用程序需要的时候，应用程序可以直接使用连接池中的连接，而不是在使用时再为应用程序建立连接，减少等待时间从而提高性能。

（2）当应用程序使用完连接时，应该将连接 close()掉，这样连接就可以返回连接池，以供下一个应用程序使用，这样同一连接可以供多个应用程序使用，节省了 CPU 等系统资源，又进一步提高了性能。

（3）缓存 Prepared Statement 和 Callable Statement。当应用程序或者 EJB 中使用 Prepared Statement 或者 Callable Statement 时，应用服务器和数据库服务器会首先对 Prepared Statement 和 Callable Statement 进行预处理，然后再进行执行。Statement cache 是 WebLogic 提供的对预处理后的 Statement 进行缓存的功能，可以减少预处理所用的时间。连接池中的每个连接都有自己的 cache，用来缓存这些 Statement 设置 Statement cache，可以到 WebLogic Console→Datasource→Connection Pool 下进行。两个参数分别是 Statement Cache Type 和 Statement Cache Size。

缓存相关的两个选项如下。

Statement Cache Type：指定缓存算法。缓存算法用来确定将哪些 Statements 进行 cache，包括 LRU（Least Recently Used）和 Fixed 两个算法。LRU 是最近最少使用算法，即当缓存满了之后，把最近最少使用的 Statement 移除缓存。Fixed 是首先用过的 Statement，直到缓存满了为止。一般情况下选择 LRU 算法。

Statement Cache Size：指有多少个 Prepared Statement 或者 Callable Statement 可以被缓存，WebLogic 在遇到对这些 Statement 的请求时会重用缓存中的 Statement，而不会重新加载，默认值为 10。

3．连接池的测试特性

（1）连接测试特性简介。

连接池的测试特性是指连接池为了保持池里的连接总是健康的，可以对池里的连接进行测试，在测试时，对数据库进行测试性操作，比如 select 1 from dual 等，以查看连接是否健康，如果不健康的话，就会重新创建该连接。

测试分为自动连接测试和手动连接测试。

① 自动连接测试是指在连接池自动对连接进行测试，具体包括下列内容。

- 定期测试未使用的连接。

- 将连接池里的连接交给应用程序使用以前，对连接进行测试。
- 在创建连接时进行测试。
- 在连接被交还回连接池时进行测试。

上面的这几个选项分别对应下面（3）中提到的 4 个参数。

注：在测试期间连接是不可用的，处于一种 reserved 状态，只有在测试成功以后，才将连接交给应用程序使用；如果测试失败，则重新打开连接，再交给应用程序使用。

② 手动连接测试，是指手动对连接进行测试，可以在 Admin Console 里执行。

（2）对哪些连接进行测试。

连接测试不是对连接池里所有的连接进行测试，根据不同的情况，会对下面的这些连接进行测试。

- 新建的连接。
- 定期测试未使用的连接。
- 测试保留的连接。
- 测试应用程序释放的连接。

（3）自动测试用到的几个参数。

Test-Frequency-Seconds/RefreshMinutes：连接容器会定期测试所有连接池里的空闲连接。使用 Test-Frequency-Seconds/RefreshMinutes 参数指定定期测试时的时间间隔。在 8.X 之前的版本用 RefreshMinutes，8.X 及之后版本用 Test-Frequency-Seconds 参数。

Test-Connections-On-Create：指定是否在创建连接时对连接进行测试，值为 true 或 false，默认是 false。

Test-Connections-On-Release：指定是否在应用程序释放连接时对连接进行测试。值为 true 或 false，默认是 false。

Test-Connections-On-Reserve：指定是否对保留的连接进行测试值为 true 或 false，默认是 false。

Seconds to Trust an Idle Pool Connection：在将连接传递到应用程序之前或定期连接测试过程期间，WebLogic Server 相信连接仍然有效并将跳过连接测试时使用连接的秒数。

上面的这些参数的用法，在后面会陆续提到。

（4）连接测试的好处与存在的问题。

① 好处：可以保证交给应用程序的连接都是健康的。

② 可能会造成的问题：降低性能，甚至导致 ResourceException 错误。

因为，如果同时进行测试的连接过多（测试期间连接不能被使用），当应用程序申请连接时，就可能没有可用连接了，就会出现 ResourceException 错误。

后续会讨论关于连接测试造成的问题。

21.2.2　数据源

JDBC 里的 Data Source 数据源，主要是用来设置通过什么 JDBC 驱动器访问哪台主机上的哪个数据库。主要是为连接池的连接指定数据源，指定要访问的数据“所在地”。可以是单个 Data Source 也可以是 Multi Data Source，使用 Multi Data Source，可以对 Data Source

进行负载均衡或者失效切换。

多数据源：多数据源是一组数据源集合作为一个虚拟数据源对外提供服务。多数据源处理服务请求的算法有两个：故障转移（Fail Over）和负载均衡（Load Balance）。其中故障转移是优先把请求发送到列表中的第一个数据源，如果第一个数据源不可用，则发给下一个数据源，以此类推直到获取到数据库连接；负载均衡是多数据源将在其成员数据源之间平均分布连接请求。在此算法中，多数据源还提供故障转移处理，即如果请求失败，该多数据源会将请求发送给列表中的下一个数据源。

备注：JDBC 中配置 Oracle 的 RAC 数据库的连接。

WebLogic 的 JDBC 中没有专门配置 RAC 连接的操作，可以配置一个单机的数据库，然后修改 domain_home 下 config/jdbc 目录下的数据库连接配置文件中，把其中连接单机数据库的 URL 修改为连接 RAC 的 URL。

21.3 与 JDBC 有关的故障

由 JDBC 引起的故障多数与连接池配置有关，主要可以分为下面几类。

（1）WLS 启动缓慢。

（2）连接池创建缓慢或失败。

（3）WLS 运行过程出现问题，比如 ResourceException、No resources available、SQL Exception 或 OutOfMemoryError 等。

（4）WLS 在运行过程中崩溃、WLS 服务器或应用程序挂起。

（5）服务器出现故障后不能重新建立连接。

（6）JDBC 不当配置导致 DB 出现问题，比如打开的游标过多等。

（7）内存泄漏。

（8）防火墙问题，比如关闭空闲连接、空闲 JMS 连接问题。

下面会分别讨论上面的这些问题。

21.3.1 由创建连接池造成的 WLS 启动缓慢

1．问题描述及成因

WLS 在启动时，会创建连接池和建立 JDBC 连接，因此如果建立 JDBC 连接时花费的时间比较多的话，WLS 的启动就会很缓慢。

2．解决办法

（1）减小 InitialCapacity 值，创建的连接少了，初始化连接池所需要的时间也就少了；使用 CapacityIncrement 值以确保单个请求，可以触发多个连接的创建。

（2）将 InitialCapacity 设置为 0，将 CapacityIncrement 设置为 MaxCapacity，将 InitialCapacity 设置为 0，然后将 CapacityIncrement 设置为 MaxCapacity。这样会在第一个

连接请求创建所有需要的连接，代价是第一个连接请求会花很长的时间，好处是在这之后，整个连接池就可以正常进行工作了。

注：某些 JDBC 驱动器，比如 Oracle 4 类驱动，在 8.1 SP2 及之后版本中使用时，可以限制连接请求的最大等待时间，对于这些驱动程序，可以在 config.xml 配置文件里配置 JDBCConnectionPool 时设置 LoginTimeout 参数。这个参数会由 WebLogic Server 传递给 JDBC 驱动器。

21.3.2　连接池创建失败

1．问题描述及成因

WebLogic Server 在启动时，会创建特定数量的、到数据库的物理连接，具体的数量由 InitialCapacity 参数来定。此时如果 JDBC 的连接配置的不正确或者数据库不可用，物理连接的初始化就会失败，连接池的创建就会失败。

在 WebLogic Server 的启动日志中，会发现类似于如下的错误。

示例 21-1：

```
<Dec 10, 2010 8:05:05 AM CST> <Warning> <JDBC> <BEA-001129> <Received
exception while creating connection for pool "mypool": The Network Adapter
could not establish the connection>
<Dec 10, 2010 8:05:06 AM CST> <Error> <Deployer> <BEA-149205> <Failed to
initialize the application 'landingbj' due to error weblogic.application.
ModuleException: .
weblogic.application.ModuleException:
       at weblogic.jdbc.module.JDBCModule.prepare(JDBCModule.java:290)
       at weblogic.application.internal.flow.ModuleListenerInvoker.
       prepare(ModuleListenerInvoker.java:93)
       at weblogic.application.internal.flow.DeploymentCallbackFlow$1.
       next(DeploymentCallbackFlow.java:387)
       at weblogic.application.utils.StateMachineDriver.nextState
       (StateMachineDriver.java:37)
       at weblogic.application.internal.flow.DeploymentCallbackFlow.
       prepare(DeploymentCallbackFlow.java:58)
       Truncated. see log file for complete stacktrace
weblogic.common.ResourceException: The Network Adapter could not establish
the connection
       at weblogic.jdbc.common.internal.XAConnectionEnvFactory.
       makeConnection(XAConnectionEnvFactory.java:472)
       at weblogic.jdbc.common.internal.XAConnectionEnvFactory.
       createResource(XAConnectionEnvFactory.java:166)
       at weblogic.common.resourcepool.ResourcePoolImpl.makeResources
       (ResourcePoolImpl.java:1149)
       at weblogic.common.resourcepool.ResourcePoolImpl.makeResources
       (ResourcePoolImpl.java:1073)
```

```
        at weblogic.common.resourcepool.ResourcePoolImpl.start(Resource-
        PoolImpl.java:232)
        Truncated. see log file for complete stacktrace
>
```

如果 WLS 启动时数据库不可用的话，WebLogic 如何处理？

（1）对于 WLS 8.1 之前的版本，将不会创建 JDBC 连接池，因此也就无法使用 JDBC 连接池。

（2）对于 WLS 8.1 和更高版本，将先创建一个大小为 0 的连接池（默认情况），如果过后 DB 变为可用，将在请求时创建连接，否则将继续报错。

2．解决办法

（1）对于 WLS 8.1 之前的版本，配置一个在启动时不会打开任何连接的连接池，也就是将 InitialCapacity 参数设置为 0。这样能成功创建连接池。之后对于每个连接请求，连接池会根据 CapaciyIncrement 参数指定的个数创建连接，当然，如果此时数据库仍然不可用，就会报错。

（2）从 8.1 的版本开始起，在创建连接池时遇到问题的话，默认就会创建一个 0 初始连接的连接池，然后等连接池配置正确或者数据库可用以后再创建连接，以供后续的应用程序使用。WebLogic 的数据库连接池中有一个参数 Connection Creation Retry Frequency 与该属性相关，Connection Creation Retry Frequency 表示重新创建数据库连接的间隔时间（以秒为单位），如果设置为 0，表示不会重连数据库。

21.3.3　JDBC 配置不正确造成的连接池创建问题

如果 JDBC 连接池配置不正确，服务器启动时连接池将创建失败。以下是几个比较常见的问题。

1．ORA-12405 错误：监听器无法识别连接描述符中指定的 SID

（1）问题描述。

Oracle ORA-12405 错误是指 Oracle 监听器无法识别客户端发送过来的连接描述符。此时常见的报错，如示例 21-2 第 4 行所示：ERR=12405。

示例 21-2：

```
<Dec 10,2010 11:45:49 PM CST>
<Warning><JDBC><BEA-001129>
<Received exception while creating connection for "mypool":Listener refused
the connection with the following error:ORA-12405, TNS:listener does not
currently know of SID given in connection descriptor  >
```

（2）解决思路。

检查 tnsnames.ora 文件中网络协议、IP、端口、SID 或 SERVICE_NAME 等参数配置

得是否正确。

2．ORA-01017 错误：无效的用户名密码

（1）问题描述。

示例 21-3：

```
<Dec 10,2010 11:40:49 PM CST>
<Error><Deployer><BEA-149255>
<Failure occurred in the execution of deployment request
With ID 1291996200335 for task 'weblogic.deploy
.configChangeTask.3'. Error is: 'weblogic.application.ModuleException: '
…
…
Weblogic.common.resourcepool.ResourceSystemException:
 Could not connect to 'oracle.jdbc.xa.client.OracleXADataSource'.
The returned message is: ORA-01017: invalid username/password; logon denied
```

如果配置的连接数据库的用户名或密码不正确或不匹配，就会出现示例 21-3 中的 ORA-01017 错误：无效的用户名密码错误。

（2）解决思路。

检查设置的用户名和密码是否正确。

3．网络适配器无法建立连接错误

（1）问题描述。

示例 21-4：

```
<Dec 10,2010 11:40:49 PM CST>
<Warning><JDBC><BEA-001129>
<Received exception while creating connection for "mypool":The Network
Adapter could not establish the connection>
```

如果 tnsnames.ora 文件或者环境变量$PATH 或$LD_LIBRARY_PATH 配置得不正确的话，就可能会遇到示例 21-4 所示的 The Network Adapter could not establish the connection：网络适配器无法连接错误。

（2）解决思路。

需要检查下列文件。

① tnsnames.ora 文件中的设置。检查 tnsnames.ora 文件中网络协议、IP、端口、SID 或 SERVICE_NAME 等参数配置是否正确。

② $PATH 和$LD_LIBRARY_PATH 环境变量。$PATH 应该包含$ORACLE_HOME/bin 目录；$LD_LIBRARY_PATH 应该包含$ORACLE_HOME/lib 和 OCI 库目录。

4．ORA-12054：服务名错误

（1）问题描述。

Oracle 错误 ORA-12054 示例如下。

示例 21-5：

```
ORA-12054: TNS could not resolve service name
```

如果 tnsnames.ora 文件或者环境变量$ORACLE_HOME 配置不正确的话，就有可能出现如示例 21-5 所示的 ORA-12054 错误。

（2）解决思路。

需要检查下列文件。

① 环境变量 $ORACLE_HOME 的设置是否正确。$ORACLE_HOME 环境变量应该为 Oracle 软件安装的根目录。

② tnsnames.ora 文件。检查 tnsnames.ora 文件中网络协议、IP、端口、SID 或 SERVICE_NAME 等参数配置是否正确。

③ 通过 sql*plus 等工具检测一下，看看是否能通过其他工具登录 Oracle 数据库；如果可以正常登录，那很可能就是配置的问题；如果不能正常登录，那么很可能是数据库的问题，此时需要对数据库做检查。

5．ORA-23326 用户认证错误

（1）问题描述。

Oracle 错误 ORA-23326 示例如下。

示例 21-6：

```
LOGIN ERROR CODE: 23326
<Error> <JDBC Connection Pool> Cannot startup connection pool "mypool"
weblogic.common.ResourceException:
Could not create pool connection. The DBMS driver exception was:
java.sql.SQLException: ORA-23326: need explicit attach before
authenticating a user.
```

如果 tnsnames.ora 文件或者环境变量$PATH 或$LD_LIBRARY_PATH 配置不正确的话，也有可能出现如示例 21-6 所示的 ORA-23326 错误。

（2）解决思路。

① tnsnames.ora 文件。对于 2 类驱动程序，需要检查 ORACLE_SID、SERVICE_NAME 和数据库服务器主机字符串值是否正确，并确保以下文件中的连接池属性配置的正确：config.xml（WLS 6.X 和更高版本）；weblogic.properties（WLS 5.X）。

例如，对于 5.X 版本，连接池服务器值必须匹配 tnsnames.ora 中的 SERVICE_NAME。

② 还要检查环境变量 $PATH 和 $LD_LIBRARY_PATH。$PATH 应该包含 $ORACLE_HOME/bin 目录；$LD_LIBRARY_PATH 应该包含$ORACLE_HOME/lib 和 OCI 库目录。

③ 尝试通过别的工具如 sql*plus 以相同身份账户登录一下，看看是否可以正常登录，如果可以正常登录，那很可能就是配置的问题；如果也不能正常登录，那么很可能是数据库的问题，此时需要对数据库做检查。

示例 21-7：

```
[oracle@vm167 ~]$ oerr ora 23326
23326, 00000, "need explicit attach before authenticating a user"
// *Cause:  A server context must be initialized before creating a session.
// *Action: Create and initialize a server handle.
```

根据提示，也有可能是数据库的问题。

6. Unable to load locale category

（1）问题描述。

错误示例如下。

示例 21-8：

```
weblogic.management.DeploymentException: Error creating connection pool
myConnectionPool:
0:Unable to load locale categories
```

如果本地环境变量设置不正确的话，可能会遇到如示例 21-8 所示的 0:Unable to load locale category 问题。

（2）解决思路。

很可能是因为本地设置的问题，此时需要检查本地环境变量。另外尝试在相同的环境下启动数据库客户端，看看是否能够正常启动，并检查环境变量的设置是否能正常起作用。

21.3.4　资源异常问题

1. 资源异常概述

（1）问题描述。

应用程序请求连接（也就是调用 getConnection()）时，可能遇到 ResourceException 资源异常错误。

（2）问题原因。

不外乎两种原因：

① 连接池中没有可用的连接；

② 没有线程可用于处理连接请求，因为 JDBC 连接使用服务器线程来执行工作，没有了线程，自然就不能正常处理连接请求了。

（3）总体排查思路。

如果遇到了 ResourceException 资源异常错误的话，需要做以下检查。

- ❑ 池中是否有足够的连接来应对暂时性的连接高峰。
- ❑ 是否有足够的连接可用于所需数量的并发客户端 JDBC 会话。
- ❑ 是否存在连接泄漏情况。
- ❑ 是否启用了自动连接测试。
- ❑ 同时保留和测试的连接是否过多。

- 连接测试是否太多频繁。
- 是否有足够的线程可用于 JDBC 工作。

2. 连接数不足造成的资源异常

（1）由于连接数确实不能满足要求造成的资源异常。

① 比如暂时性的连接高峰，或者所需的并发 JDBC 会话数量增多。

② 池中的连接数受限于 MaxCapacity 所设置的最大值；默认值为默认的执行线程数，对于开发模式默认为 15，对于生产模式默认为 24。

③ 所需池的大小。实际池的大小可能会更大，但适当设置 MaxCapacity 可以将并行 DB 访问控制在 DB、网络、CPU 等的容量限制之内。

（2）连接泄漏造成的可用连接数过少

① 首先什么是连接泄漏？

应用程序在使用完连接以后，应该调用 close()“关闭”该连接（此处的关闭是指释放），将连接返还回连接池，以便该连接可供其他应用程序使用，如果应用程序没有释放连接，从而导致该连接不能被其他应用程序使用，这种现象就是连接泄漏。如果有太多的连接处于泄漏状态的话，那么连接池中的可用连接将会越来越少，这样最终很可能导致出现 ResourceException 资源异常错误。

症状示例如下。

示例 21-9：

```
<BEA-101017>  SQL state[null];error code[0];
weblogic.common.resourcepool.ResourceLimitException:
No resources currently available in pool claimDataSourceTest to allocate
to applications,please increase the size of the pool and retry
```

② 解决方法。

对于连接泄漏，WLS 提供了一种检测连接泄漏的功能，可以在 config.xml 文件中将 ConnLeakProfilingEnabled 设置为 true。也可以在 admin console 里进行设置。在开启检测功能以后，在从连接池里分配了连接对象并将其赋给某个客户端以后，连接池就会开始保存堆栈跟踪信息，当发现出现连接泄漏时，就会报告相关的堆栈跟踪信息。连接池如何判断连接泄漏？当发现之前分配的连接对象，被当做垃圾回收了，就判定该连接出现了连接泄漏。另外，可以对产生的堆栈跟踪进行分析以辅助查找泄漏的具体原因。

注：连接泄漏探测功能会占用额外的资源，很可能降低连接池的性能，所以不建议在生产环境中使用此方法。

③ 对于 Oracle 数据库，在出现连接泄漏以后，很可能会遇到：

```
"ORA-00020 - maximum number of processes (600) exceeded"
```

（3）自动连接测试造成的可用连接数过少。

① 问题成因。

在进行连接测试时，不管是手动还是自动都会将连接处于一种 reserved 的状态，在测

试期间连接不能被应用程序使用。此时，如果设置了自动测试，那么在某一时刻就可能会出现：有太多的连接处于 reserved 状态，连接池里没有空闲连接。如果此时又有应用程序申请连接，那么就会报 ResourceException 错误。

如果在进行排查时，通过分析应用程序，发现连接池里保留的连接数要比应用程序使用的连接数多，并且也已经排除了连接泄漏的可能，那么很有可能是因为自动测试造成的连接数不够的问题。

② 解决方法。

将时间间隔设置得小一些，设置 Test-Frequency-Seconds/Refresh-Minutes 参数。

考虑禁用自动连接测试。默认就是禁用的。

对于不同的 WLS 版本，连接测试的具体情况不一样，如下所述。

a. 对于 WLS 7.X 和更低版本的连接测试。自动连接测试会定期捕获和测试所有未被使用的连接，测试频率由 Refresh-Minutes 参数定义。如果池测试进行得过于频繁，就可能会导致 ResourceException 错误出现；可以将 Refresh-Minutes 间隔值设置得较高或者设置为 99999999，以达到禁用自动测试的效果。

b. 对于 8.X 及更高版本的连接测试。8.X 及更高版本的 WLS 已经不再支持 Refresh-Minutes 参数，因此应该设置为 0。未使用连接的测试频率由 Test-Frequency-Seconds 参数定义，同时测试的未使用连接最大数量由 HighestNumUnavailable 参数定义。

21.3.5　ORA-01000 打开的游标数过多错误

1．问题描述

示例 21-10：

```
Java.sql.SQLException:
ORA-00604: error occurred at recursive SQL level 1
ORA-01000: maximum open cursors exceeded
```

如示例 21-10 所示，如果遇到了 Oracle ORA-01000 的 SQL Exception 错误的话，表示会话打开的游标已经达到数据库允许单个会话同时打开的游标的上限。

什么对象使用游标呢？

（1）prepared statement 对象。

（2）其他对象，如存储过程或结果集。

注：虽然，每个 prepared statement 使用一个打开的游标，但是考虑到每个连接会缓存 StatementCacheSize 个 prepared statement，而连接池里最多又可能有 MaxCapacity 个连接，那么最终可能就会打开 MaxCapacity * StatementCacheSize 个游标。也就是说，prepared statement 对象可能会需要 MaxCapacity * StatementCacheSize 个游标。

2．解决方法

（1）检查数据库是否配置了足够多的游标来满足所有池及其他对象的要求。每个池所

需要的游标值为 MaxCapacity * StatementCacheSize+备用游标。Oracle 数据库中，控制每个会话可以一次打开的游标的个数的参数是 open_cursors（9i，10g，11g）。

（2）更改数据库的限制，增大允许打开的游标的上限。因为也确实存在游标不够用的问题。

如果是 Oracle 数据库的话，可以 alter system set open_cursors=n scope=both。

（3）Oracle 的某些驱动存在游标泄漏问题。

某些版本的 Oracle 驱动（thin 或 oci）在 XA 驱动类（oracle.jdbc.xa.client. OracleXA-DataSource）中存在游标泄漏问题，所以在一定时间过后，就可能造成打开游标过多的错误。

此时，需要确保在数据库中，DBA_PENDING_TRANSACTION 视图有正确的权限。

示例 21-11：

```
grant DBA_PENDING_TRANSACTIONS to public
grant DBA_PENDING_NEIGHBORS to public
grant DBA_2PC_PENDING to public
```

注：游标泄漏问题，在 Oracle 数据库 9.2.0.5 及以后版本中得到了解决。

21.3.6 ORA-03113 连接中断错误或 01012 未登录错误

ORA-03113 错误示例如下。

示例 21-12：

```
<Jan 31, 2002 2:20:17 PM PST> <Info> <JDBC Pool oraclePool> <null> <This
connection will now be refreshed.>
<Jan 31, 2002 2:20:18 PM PST> <Info> <JDBC> <001067> <Connection for pool
"oraclePool" refreshed.>
<Jan 31, 2002 2:20:18 PM PST> <Info> <JDBC Pool oraclePool> <null> <A
connection from pool oraclePool was tested during reserve with a select
count(*) from dual and failed:>
<Jan 31, 2002 2:20:18 PM PST> <Info> <JDBC Pool oraclePool> <null>
<java.sql.SQLException: ORA-03113: end-of-file on communication channel
at weblogic.db.oci.OciCursor.getCDAException(OciCursor.java:230)
at weblogic.jdbc.oci.Statement.execute(Statement.java:534)
at weblogic.jdbc.common.internal.ConnectionEnv.test(ConnectionEnv.
java:961)
at weblogic.jdbc.common.internal.ResourceAllocator.reserve(Resource
Allocator.java:651)
at weblogic.jdbc.common.internal.ResourceAllocator.reserveUnused
(ResourceAllocator.java:575)
. . .
```

1．问题描述

如果数据库出现了间歇性故障，并且又将 TestConnectionsOnReserve 参数设置为 true（在连接交给应用程序之前先测试属性），那些在数据库故障期间出现异常的连接就会被刷新，此时在 WebLogic 的日志文件里，就会有相应的记录，类似如下：

```
ORA-03113 end-of-file on communication channel and/or ORA-01012 not logged on
```

2．解决方法

对于数据库间歇性故障，为确保连接池可以从 DB 失败中恢复，可以启用连接自动测试和重建。

21.3.7　防火墙关闭空闲连接问题

1．问题描述

如果在 WLS 和数据库之间使用了防火墙的话，在连接空闲一定长时间之后，还没有被使用的话，防火墙就可能会关闭这些连接。但是，对于这些连接，WLS 和数据库却都认为是好的，是可用的，所以就会去尝试写这些 Socket 连接，连接又已经被关闭，此时就会报错。Oracle 的 ORA-03113 错误是其典型的故障症状。

ORA-03113 错误示例如下。

示例 21-13：

```
java.sql.SQLException: ORA-03113: end-of-file on communication channel
at weblogic.db.oci.OciCursor.getCDAException(OciCursor.java:230)
at weblogic.jdbc.oci.Statement.executeQuery(Statement.java:916)
at ...
```

2．解决方法

使用连接池的自动测试或者刷新功能，设置下面两个参数之一，确保在防火墙超时期间内每个连接至少被激活一次。

RefreshMinutes（8.X 之前的版本使用）。

TestFrequencySeconds（8.X 及之后的版本使用）。

设置了上面的参数后，每隔一定时间就会对连接进行测试，测试时，就会通过连接和数据库进行交互，就会使连接处于非空闲状态，这样就能限制最长空闲时间，使其保持在防火墙的判断标准范围内。

注：自动连接测试可能会带来的问题上面有相关说明，在设置时，需要根据实际需要综合考虑。

21.3.8 防火墙关闭空闲 JMS 连接问题

1. 问题描述

如果 JMS 采用 JDBC 持久性存储，每个 JMS 服务器会使用 JDBC 连接池中的一个连接，由于该连接是保留连接，因此不会被视为空闲连接，该 JMS 连接也就不会得到定期测试，因此，如果 JMS 连接长期空闲的话，防火墙就可能会关闭该 JMS 服务器连接，异常便由此而生。

JMS 连接问题异常示例如下。

示例 21-14：

```
JMSServer "myJMSServer", store failure while writing message for queue
myQueue, java.io.IOException
```

2. 解决方法

要避免防火墙关闭 JMS 服务器的 JDBC 连接，可以使用下列方法。

（1）在防火墙超时期间内至少发送一个伪消息。

（2）禁用防火墙的关闭连接功能。

（3）定义一个单独的 JMS 服务器 JDBC 池，并使用 weblogic.Admin RESET_POOL 命令，在每个超时期间内至少再打开一次连接。

（4）使用 WLS 6.1 SP7 或者更高版本，这些版本会每 5 分钟对空闲 JMS 连接执行一次 ping 命令，以防止防火墙关闭连接。

21.3.9 WebLogic Server 崩溃

1. 问题描述

2 类 JDBC 驱动程序可以将标准 JDBC（Java）方法调用转换为本地数据库 API 调用，而本地调用的问题可能会造成 JVM 和 WLS 的崩溃。有关排除服务器崩溃故障的信息，可以参考“二进制核心文件分析模式”，里面有较详细的介绍。

另外，JDBC 连接使用 WebLogic Server 执行线程来执行它的工作。如果出现下列问题，服务器线程可能会挂起，例如，需要重建大量连接。或者数据库基础结构问题或死锁也会导致请求挂起，其结果可能是应用程序挂起或整个服务器挂起。

有关排除服务器挂起故障的信息可以参考“常规服务器挂起模式”，里面有详细的介绍。

2. 解决方法

如果确定是由于这些本地调用造成的 WLS 崩溃或者 WLS 或应用程序挂起引起的，可以采用的方法如下。

（1）改为使用纯 Java（4 类）JDBC 驱动程序。

（2）进行记录和调试可能会确定故障发生的原因。

21.3.10　内存泄漏故障

1．问题描述

JDBC 对象可以在本地内存或 JVM 堆的内存中创建，具体视所使用的驱动程序而定。无论在何处创建，只要出现下列情况，就可能会发生内存泄漏。

（1）未正确关闭连接。

（2）未正确释放对象。

（3）内存泄漏会导致内存不足问题，例如，OutOfMemoryError 异常。

2．解决方法

有关排除内存泄漏故障的信息可以参考“内存不足和内存泄漏故障”，里面有详细的排查及解决方法。

21.3.11　连接被重建的问题

1．问题描述

某些程序需要使用数据库驱动提供的专有特性（非标准 JDBC 特性），此时就需要使用 getVendorConnection()方法来获取数据库驱动提供的 Connection 类实例。但是默认情况下，如果调用 getVendorConnection()然后应用程序释放该连接，JDBC 连接池为了确保连接的安全与可用性，会自动重建连接，这就使得连接池的优点显著，即通过重用连接来降低系统开销，无法得到实现。

2．解决方法

禁用连接重建。

连接池是否重建连接受参数 RemoveInfectedConnectionsEnabled 的影响，该参数默认值为 true，如果要禁用的话，可以将其设置为 false。

注：这里需要根据您具体的程序而定，某些情况下可能确实需要对连接进行重建。

21.4　针对生产环境中 JDBC 的调整建议

（1）将初始池大小和最大池大小设置为相等，即设置 InitialCapacity=MaximumCapacity，以确保在创建连接池时就打开所有连接。

因为创建一个到物理数据库连接可能是一个开销较大的操作，所以应该在创建连接池时，就创建所有的连接，并将这些连接保持为打开的状态。

（2）将连接数设置为至少等于或者大于使用连接的执行线程的数量。这有助于避免

ResourceExceptions 错误。

（3）不需要时，禁用连接测试。

对连接进行测试也是一种开销，应该尽可能地避免。如果设置了在使用前进行测试（TestConnectionsOnReserve）的话，就没必要在连接被归还回连接池时也进行测试了。TestConnectionsOnRelease 默认是 false，如果您设置为 true 的话，则需要考虑一下是否真的需要。

（4）禁用池收缩，即将 ShrinkageEnabled 设置为 false。

什么是连接池收缩？在某些情况下，比如连接高峰过去以后，可能在一段时间之内连接池里有一些连接一直处于空闲的状态，如果此时 WLS 整体资源紧张，这也是一种资源的闲置，所以默认连接池会每隔一定的时间减少连接池里连接的数量。减少到的个数需要看 InitialCapacity 和当前正在使用的连接的个数哪个大，哪个大减小到哪个。同样由于创建连接是个开销比较大的操作，如果不需要的话，应该禁用连接池收缩。

21.5 故障排除检查清单

21.5.1 故障排除策略

首先，尽可能多的收集问题的有关信息。确保很好地理解了，应用程序是如何使用 JDBC 的，并且需要确认正确的按照应用程序的要求，对 JDBC 进行配置。

（1）探查应用程序对 JDBC 的使用方式，并确认 JDBC 配置反映了该使用方式。

- 同时使用的连接数。
- 是否在启动时创建所有连接。
- 使用启用了自动连接测试及其执行效率。
- 应用程序是否使用需要直接访问连接对象的高级 JDBC 功能。

（2）根据故障症状（如下列故障症状）确定问题性质。

- 池创建是否在服务器启动时失败。
- 是否因资源问题而记录了异常。
- 是否发生连接泄漏。

（3）解决具体的故障原因如下。

- 更正任何配置错误。
- 调整连接测试策略。
- 更正导致连接泄漏的应用程序代码。

21.5.2 其他故障排除策略

启用 JDBC 调试来收集更多信息。如果常规的方法还是不能解决遇到的问题的话，可以考虑启用 JDBC 调试来收集信息，并分析解决问题。

（1）启用 JDBC 记录功能，即通过管理控制台或 config.xml 设置 JDBCLoggingEnabled。

（2）在 ServerDebugMBean 上启用特定调试区域标志，包括 JDBCConn、JDBCSQL 和 JTAJDBC。

① 可以在 config.xml 文件里打开。

示例 21-15：

```
<Server Name="myserver" > . . .
      <ServerDebug Name="myserver" JDBCConn="true"
                JDBCSQL="true" JTAJDBC="true" />
</Server>
```

② 也可以在 WLS 启动时设置，或者在 WLS 启动时设置。

示例 21-16：

```
-Dweblogic.Debug=weblogic.JDBCConn,weblogic.JDBCSQL,weblogic.JTAJDBC
```

注：JDBC 是一个多层次的子系统，只有一部分运行在 WLS 里，所以对 JDBC 的调试，很大程度上跟 JDBC 驱动有关，更多关于 JDBC 驱动的调试和追踪信息需要从驱动提供商那里获得。上面的这些调试标志只是 WLS 提供的。

（3）对于不允许使用 JDBC 调试或者打印的信息不够的话，可以考虑安装 P6Spy 驱动，P6Spy 驱动可以帮助调试在 JDBC 和 DB 之间传送的 SQL 语句。

P6Spy 驱动可以去 http://www.p6spy.com/ 下载。

P6Spy 的相关文档，可以参考 http://www.p6spy.com/documentation/index.htm。

注：跟踪输出信息可能会非常详细，所以需要仔细考虑需要使用哪些标志。另外，跟踪调试可能会影响系统的性能。所以不要在生产环境中做追踪或调试。

第 22 章　全局事务与 JTA 的支持故障

22.1　什么是分布式事务与全局事务

22.1.1　事务及事务操作 ACID 特性

事务（Transaction）又称为交易，是指在一个或多个资源，如数据库或文件上，为完成某个功能执行过程的集合，作为一个完整的事务，应该保证在一个事务中所有操作的 ACID 特性，具体如下。

- **Atomicity（原子性）**　一个事务中的所有操作全部发生或一个也不发生。
- **Consistency（一致性）**　事务的完成必须使系统保持一致的状态。事务只是一个工具，使一致性保证成为可能，而它本身并不是一致性的保证者。
- **Isolation（隔离性）**　正在执行的事务不应彼此影响。一个事务的参加者应该只能看到自己事务中操作的中间状态，而不是其他事务的中间状态。
- **Durability（永久性）**　一个成功的交易其结果是不能改变的，除非有另外一个交易来改变它。

我们可以把去银行取钱看作一个事务，简化一下：为了完成存钱这一功能，需要两个过程，一个是银行工作人员把您账户的钱减少，另外一个是银行工作人员把钱给您。

Atomicity 特性是指要么银行工作人员把您账户的钱减少，并把钱交给您；要么您没有去取钱，当然银行工作人员也不会把您账户上的钱减少。不存在您没有去取钱，银行工作人员把您账户上的钱减少这一情况，也不存在您取了钱之后，账户上的钱没有变化这一情况。

Consistency 是指再假定银行规定账户上的钱不允许有负值。那么在您去银行取钱之前和取钱之后，您账户上的余额都不会是负数。

Isolation 特性是指您在银行取钱的时候，会有很多人在银行取钱或者存钱。别人取钱或者存钱肯定不会影响到您取钱这一事物，既不会影响到您取出钱的多少，也不会影响到您账户余额的变化。

Durability 是指您在银行取钱成功之后，结果是您的银行账户上余额减少。除非您继续往该账户上存钱或者取钱，否则银行账户上的余额不会变更。

22.1.2　分布式事务处理

分布式事务的概念：分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。分布式事务也是事务的一种，

符合事务操作的 ACID 特性。

分布式事务处理是指一个事务可能涉及多个数据库操作，分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生一致的结果（全部提交或全部回滚）。

通常把一个数据库内部的事务处理，如对多个表的操作作为本地事务看待。数据库的事务处理对象是本地事务，而分布式事务处理的对象是全局事务。

22.1.3　全局事务

所谓全局事务，是指分布式事务处理环境中，多个数据库可能需要共同完成一个工作，这个工作即是一个全局事务，例如，一个事务中可能更新几个不同的数据库。对数据库的操作发生在系统的各处，但必须全部被提交或回滚。此时一个数据库对自己内部所做操作的提交不仅依赖本身操作是否成功，还要依赖于全局事务相关的其他数据库的操作是否成功，如果任一数据库的任一操作失败，则参与此事务的所有数据库所做的所有操作都必须回滚。

一般情况下，某一数据库无法知道其他数据库在做什么，因此，在一个 DTP（Distributed Transaction Processing，分布式事务处理）环境中，交易中间件是必需的，由它通知和协调相关数据库的提交或回滚。而一个数据库只将自己所做的操作（可恢复）映射到全局事务中。

22.1.4　XA 与两阶段提交协议

X/Open 组织（即现在的 Open Group）定义了分布式事务处理模型。X/Open DTP 模型（1994）包括应用程序（AP）、事务管理器（TM）、资源管理器（RM）、通信资源管理器（CRM）4 部分。一般，常见的事务管理器（TM）是交易中间件，常见的资源管理器（RM）是数据库，常见的通信资源管理器（CRM）是消息中间件。

XA 就是 X/Open DTP 定义的交易中间件与数据库之间的接口规范（即接口函数），交易中间件用它来通知数据库事务的开始、结束以及提交、回滚等。XA 接口函数由数据库厂商提供。

通常情况下，交易中间件与数据库通过 XA 接口规范使用两阶段提交来完成一个全局事务，XA 规范的基础是两阶段提交协议。

在第一阶段，交易中间件请求所有相关数据库准备提交（预提交）各自的事务分支，以确认是否所有相关数据库都可以提交各自的事务分支。当某一数据库收到预提交后，如果可以提交属于自己的事务分支，则将自己在该事务分支中所做的操作固定记录下来，并给交易中间件一个同意提交的应答，此时数据库将不能再在该事务分支中加入任何操作，但此时数据库并没有真正提交该事务，数据库对共享资源的操作还未释放(处于上锁状态)。如果由于某种原因数据库无法提交属于自己的事务分支，它将回滚自己的所有操作，释放对共享资源上的锁，并返回给交易中间件一个失败应答。

在第二阶段，交易中间件审查所有数据库返回的预提交结果，如所有数据库都可以提

交，交易中间件将要求所有数据库做正式提交，这样该全局事务被提交。而如果有任一数据库预提交返回失败，交易中间件将要求所有其他数据库回滚其操作，这样该全局事务被回滚，如图 22-1 所示。

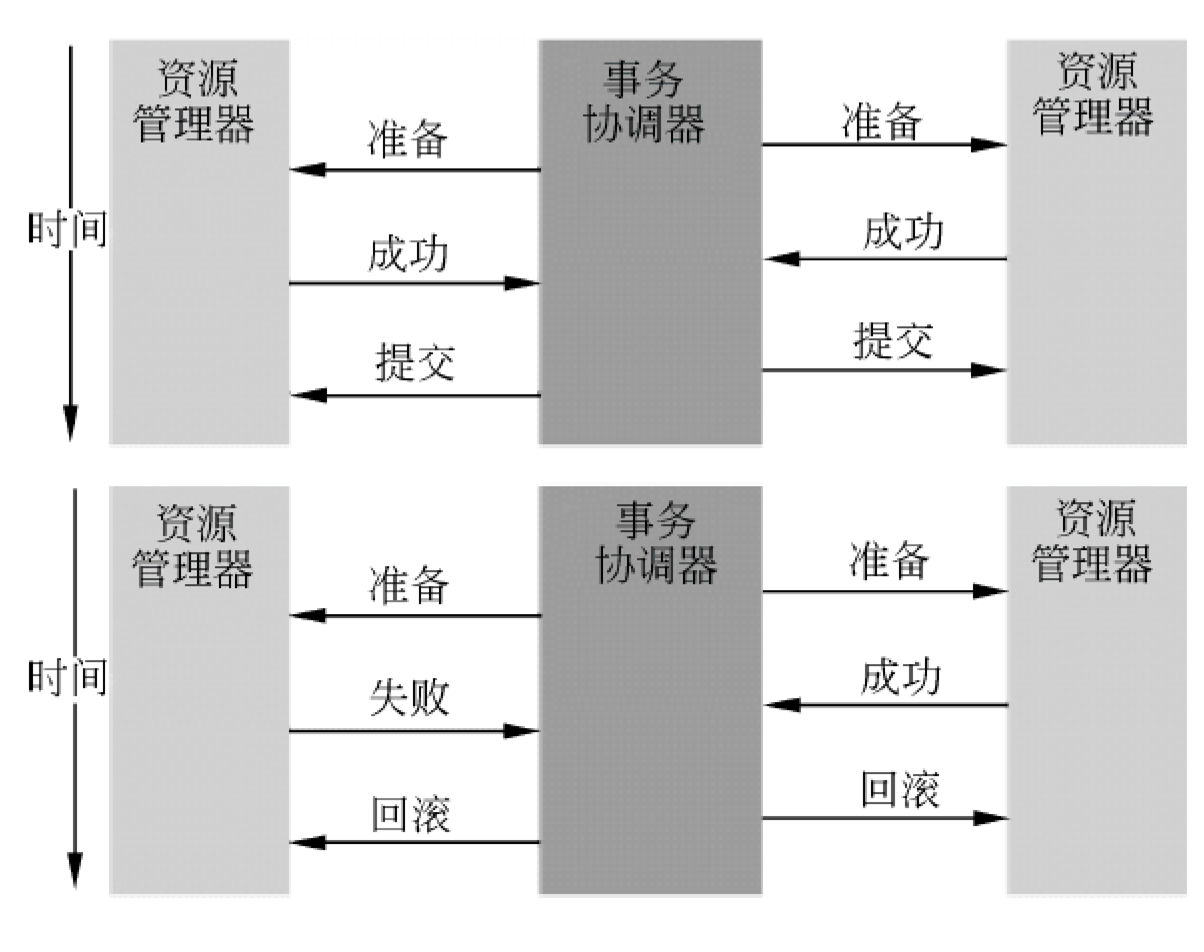


图 22-1

XA 规范对应用来说，最大的好处在于事务的完整性由交易中间件和数据库通过 XA 接口控制，AP 只需要关注与数据库的应用逻辑的处理，而无需过多关心事务的完整性，应用设计开发会简化很多。

22.2 如何使用全局事务

22.2.1 配置连接池和数据源

WebLogic 提供数据库连接池。所谓连接池，是在 WebLogic 服务器端提供的与数据库连接的池，这个池可以在 WebLogic 启动时或运行中时建立，并且可以根据运行情况进行伸缩。建议采用 WebLogic 连接池，主要通过 WebLogic 控制台进行连接池管理。

在 WebLogic 控制台中，在主控制菜单执行 Connection Pools 命令，接着执行 Configure a new JDBC Connection Pool（创建一个新的 JDBC 连接池）命令，然后在创建页面中输入有关参数。

数据源提供了一个应用程序与数据库连接池之间的接口。应用程序可以通过数据源操作数据库。WebLogic 提供两种数据源类：一个为普通 DataSource，另一种为 TxDataSource，支持分布式事务处理。在基于 WebLogic 的应用中，数据源是主要的数据库访问手段。

通过控制台可以建立数据源。在建立数据源之前，应该首先建立相关的连接池。

WebLogic 中的数据库事务操作需要采用数据源 DataSource，尤其是 TxDataSource。因此，在使用事务处理之前，需要建立一个 TxDataSource。

22.2.2　为数据源配置事务选项

JDBC 数据源的事务协议确定在事务处理期间如何处理数据源中的连接。

要为 JDBC 数据源配置事务选项，可执行下列操作。

（1）如果尚未执行此操作，可在管理控制台的更改中心中单击“锁定并编辑”按钮。

（2）在“域结构”树中，展开“服务”→“JDBC”子树，然后选择“数据源”选项。

（3）在“数据源概要”页上，单击数据源名。

（4）选择“配置：事务”选项卡。

（5）选择“全局事务”选项。

选中“支持全局事务”复选框（默认）将在此数据源中启用全局事务支持。清除此复选框将在此数据源中禁用（忽略）全局事务。大多数情况下，应选中此选项。

如果已选中“支持全局事务”复选框，请选择进行事务处理的一个选项。

- **记录上一个资源**　选择此选项启用非 XA JDBC 连接，可以通过使用“记录上一个资源”（LLR）事务优化来参与全局事务。建议您用此选项代替“仿真两阶段提交”。
- **仿真两阶段提交**　启用非 XA JDBC 连接可以通过使用 JTA 来仿真分布式事务中的参与。只有在应用程序可允许出现试探性情况时才能选择此选项。
- **一阶段提交**　选择此选项启用非 XA 连接，从而作为唯一的事务参与者可以参与全局事务。

（6）单击“保存”按钮。

（7）要激活这些更改，可在管理控制台的更改中心中单击“激活更改”按钮。并非所有更改都立即生效，某些更改必须重新启动后才能生效。

> **注意**
> 如果数据源使用 XA JDBC 驱动程序创建数据库连接，数据源中的连接将仅支持两阶段提交事务协议。其他任何事务选项对于使用 XA JDBC 驱动程序的数据源均不可用。

22.2.3　全局事务样例

为 Java 程序提供分布式事务开发的接口是 JTA（Java Transaction API），要用 JTA 进行事务界定，应用程序需要调用 javax.transaction.UserTransaction 接口中的方法。用 JTA 界定事务，那么就需要有一个实现 javax.sql.XADataSource、javax.sql.XAConnection 和 javax.sql.XAResource 接口的 JDBC 驱动程序。一个实现了这些接口的驱动程序将可以参与 JTA 事务。一个 XADataSource 对象就是一个 XAConnection 对象的工厂。XAConnection 是参与 JTA 事务的 JDBC 连接。

要使用 JTA 事务，必须使用 XADataSource 来产生数据库连接，产生的连接为一个 XA 连接。

JTA 方式的实现过程如下。

用 XADataSource 产生的 XAConnection 扩展了一个 getXAResource()方法，事务通过这个方法把它加入到事务容器中进行管理。对于调用者来说，根本看不到事务是如何管理的，

您只要声明开始事务，告诉容器下面的操作要求事务参与了，最后告诉事务说到这儿可以提交或回滚了，别的都是黑箱操作。

在使用 JTA 之前，您必须首先实现一个 Xid 类用来标识事务（在普通情况下这将由事务管理程序来处理）。Xid 包含 3 个元素：formatID、gtrid（全局事务标识符）和 bqual（分支修饰词标识符）。

（1）实现自己的 Xid。

示例 22-1：

```
import javax.transaction.xa.*;

public class MyXid implements Xid {

      // Xid 包含的 3 个元素
protected int formatid;
protected byte gtrid[ ];
protected byte bqual[ ];

public MyXid() {

}

  public MyXid(int formatid, byte gtrid[ ], byte bqual[ ]) {
   this.formatid = formatid;
   this.gtrid = gtrid;
   this.bqual = bqual;
  }

public int getFormatid() {
   return formatid;
  }

  public byte[ ] getBranchqualifier() {
   return bqual;
  }

  public byte[ ] getGlobaltransactionid() {
   return gtrid;
  }
}
```

（2）通过 jndi 找到 WebLogic 中配置好的 XA 数据源。

示例 22-2：

```
public static XADataSource getXADataSource() throws Exception {
```

```
InitialContext ctx = new InitialContext( mgr.getprops());
XADataSource ds = (XADataSource)ctx.lookup("jdbc/xads");
return ds;
}
```

（3）使用 xadatasource 得到 xaconnection，使用 xaconnection 得到 xaresource，基于 xaresource 进行具体的数据访问。如果这里的 lookup 得到多个 xadatasource，然后得到多个 xaresource，就可以实现多数据源的事务控制了。

示例 22-3：

```
XADataSource xaDS;
XAConnection xaConn;
XAResource xaRes;
Xid xid;
Connection conn;
Statement stmt;
int ret;
xaDS = getXADataSource ();
xaConn = xaDS. getXAConnection ();
xaRes = xaConn. getXAResource ();
conn = xaRes.getConnection();
stmt = conn.createStatement();
xid = new MyXid(100, new byte[ ]{0x01}, new byte[ ]{0x02});

try {
   xaRes.start(xid, XAResource.TMNOFLAGS);
   stmt.executeUpdate("insert into test_table values (100)");
   stmt.executeUpdate("insert into test_table values (50)");
   xaRes.end(xid, XAResource.TMSUCCESS);
   ret = xaRes.prepare(xid);

   if (ret == XAResource.XA_OK) {
   xaRes.commit(xid, false);
}

}
   catch (XAException e) {
       e.printstacktrace();
       xaRes.rollback(xid);
}
   finally {
   stmt.close();
   conn.close();
   xaConn.close();
}
```

上述例子在客户端来进行全局事务的管理和控制。其中涉及到的数据库操作都需要通过 TxDataSource 来获取连接对象。这样，通过这个连接对象所完成的操作都会包含在此全局事务中。

JDBC 事务（不支持分布式事务处理）。在 JDBC 中怎样将多个 SQL 语句组合成一个事务呢？在 JDBC 中，打开一个连接对象 Connection 时，缺省是 auto-commit 模式，每个 SQL 语句都被当做一个事务，即每次执行一个语句，都会自动得到事务确认。为了能将多个 SQL 语句组合成一个事务，要将 auto-commit 模式屏蔽掉。在 auto-commit 模式屏蔽掉之后，如果不调用 commit()方法，SQL 语句不会得到事务确认。在最近一次 commit()方法调用之后，所有的 SQL 会在方法 commit()调用时得到确认。

代码如下。

示例 22-4：

```
try{
    Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
    String url="jdbc:oracle:thin:@localhost:1520:orcl";
    String user="test";
    String password="test";
    Connection conn= DriverManager.getConnection(url,user,password);
    conn.setAutoCommit(false);        //把 auto-commit 模式屏蔽掉
    stmt1= conn. createStatement();
    stmt2= conn.createStatement();
    String sql1 = " insert into test_table values (100)";
    String sql2 = " insert into test table values (50)";
    stmt1.executeUpdate(sql1);
    stmt2.executeUpdate(sql2);

    conn.commit();        //两个 SQL 作为一个事务提交
    }catch(SQLException e){
      conn.rollback();
      e.printStackTrace();
    } finally{
      if(stmt2!=null){
       stmt2.close();
      }
    if(stmt1!=null){
      stmt1.close();
    }
    if(conn!=null){
      conn.close();
    }
}
```

JTA 提供了跨数据库连接（或其他 JTA 资源）的事务管理能力。这一点是与 JDBC Transaction 最大的差异。JDBC 事务由 Connection 管理，也就是说，事务管理实际上是在

JDBC Connection 中实现的。事务周期限于 Connection 的生命周期。

JTA 事务管理则由 JTA 容器实现，JTA 容器对当前加入事务的众多 Connection 进行调度，实现其事务性要求。JTA 的事务周期可横跨多个 JDBC Connection 的生命周期。

22.3　相关的 WebLogic 中 JTA 设置问题

WebLogic 提供对事务管理的全面支持。它的事务管理不仅可以完成对于数据库资源的管理控制，也可以完成对 JMS 的管理控制。在一个事务中，可以同时对多个数据库操作以及对多个 JMS 资源操作，它们都可以通过 JTA 接口进行操作。

配置了 WebLogic JTA 以及任何事务参与内容之后，系统就可以使用 JTA API 和 WebLogic JTA 扩展管理事务了。

在对数据库操作中，首先应该保证相关的数据库驱动程序支持两阶段提交和分布式事务处理。请注意下列事项。

- ❑ JTA（事务）的配置设置可在域级别应用。这就意味着配置特性设置将应用于域中的所有服务器中。
- ❑ JTA 的监视任务是在服务器级别执行的。在管理控制台中，可以监视域中每个服务器的事务。事务统计信息针对特定服务器而显示，而不针对整个域。
- ❑ 参与资源（如 JDBC 数据源）的配置设置是基于每个配置的对象的。这些设置应用于特定对象的所有实例。

第 23 章　中文乱码相关问题

23.1　引言

先来看看几种常见的编码方式。

Unicode：Unicode.org 制定的编码机制，能将全世界的常用文字都囊括进去，在 Unicode 2.0 开始抛弃了 16 位限制，原来的 16 位作为基本位平面，另外增加了 16 个位平面，相当于 20 位编码，编码范围 0 到 0x10FFFF。

UCS：ISO 制定的 ISO 10646 标准所定义的 Universal Character Set，采用 4byte 编码。

UTF-8：8bit 编码，ASCII 不做变换，其他字符做变长编码，每个字符 1～3 byte，通常作为外码，有以下优点。

（1）与 CPU 字节顺序无关，可以在不同平台之间交流。

（2）容错能力高，任何一个字节损坏后，最多只会导致一个编码码位损失，不会连锁错误（如 GB 码错一个字节就会整行乱码）。

GB2212：简体中文的编码

GBK：对 GB2212 的扩充，以容纳 GB2212 字符集范围以外的 Unicode 2.1 的统一汉字部分，并且增加了部分 Unicode 中没有的字符，支持简体中文及繁体中文。

GBK、GB2212 等与 UTF8 之间都必须通过 Unicode 编码才能相互转换。从节省空间的角度考虑，中文内容多的站点可以考虑用 GBK 或 GB2212，英文内容多的站点还是 UTF-8 好。

Java、JSP 源文件中很可能包含有中文，而 Java 和 JSP 源文件的保存方式是基于字节流的，如果 Java 和 JSP 编译成 Class 文件过程中，使用的编码方式与源文件的编码不一致，就会出现乱码。基于这种乱码，在 Java 文件中尽量不要写中文（注释部分不参与编译，写中文没关系），如果必须写的话，尽量手动带参数-ecoding GBK 或-ecoding GB2212 进行编译；对于 JSP，相对还好处理，在文件头加上一些设置基本上就能解决这类乱码问题。很多存储媒介，如数据库、文件、流等的存储方式都是基于字节流的，Java 程序与这些媒介交互时就会发生字符（char）与字节（byte）之间的转换，具体情况如下。

从页面 form 提交数据到 Java 程序 byte→char。

从 Java 程序到页面显示 char→byte。

从数据库到 Java 程序 byte→char。

从 Java 程序到数据库 char→byte。

从文件到 Java 程序 byte→char。

从 Java 程序到文件 char→byte。

从流到 Java 程序 byte→char。

从 Java 程序到流 char→byte。

如果在以上转换过程中使用的编码方式与字节原有的编码不一致，很可能就会出现乱码。解决这些乱码问题的关键在于确保转换时使用的编码方式与字节原有的编码方式保持一致。简单起见，建议在系统中尽量全部采用 UTF-8 编码。

23.2　JSP 与页面参数之间的乱码

JSP 页面可以用下面语句来设置编码：<%@ page contentType="text/html;charset=gbk" %>、<%@ page pageEncoding="gbk"%>。JSP 获取页面参数时一般采用系统默认的编码方式，如果页面参数的编码类型和系统默认的编码类型不一致，很可能就会出现乱码。解决这类乱码问题的基本方法是在页面获取参数之前，强制指定 request 获取参数的编码方式：request.setCharacterEncoding("GBK") 或 request.setCharacterEncoding("GB2212")。如果在 JSP 将变量输出到页面时出现了乱码，可以通过设置 response.setContentType ("text/html;charset=GB2212")来解决。如果不想在每个文件里都写这样两句话，更简洁的办法是使用 Servlet 规范中的过滤器指定编码，具体如下。

首先编写 Filter 类。

示例 23-1：

```
package myFilter;
import java.io.IOException;
import javax.servlet.*;
public class ChangeCharsetFilter implements Filter {
    protected String encoding = null;/////要制定的编码，在 web.xml 中配置
    protected FilterConfig filterConfig = null;
    public void destroy() {
        this.encoding = null;
        this.filterConfig = null;
    }
    public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain)throws IOException, ServletException {
        if (request.getCharacterEncoding() == null){
            String encoding = getEncoding();////得到指定的编码名字
            if (encoding != null)
                request.setCharacterEncoding(encoding);////设置 request 的编码
        }
         chain.doFilter(request, response);///有机会执行下一个 Filter
    }
    public void init(FilterConfig filterConfig) throws ServletException {
         this.filterConfig = filterConfig;
         this.encoding = filterConfig.getInitParameter("encoding");//得到在
         web.xml 中配置的编码
```

```
        }
        protected String getEncoding() {
            return (this.encoding);///得到指定的编码
        }
    }

    ※编辑 web.xml 文件，添加如下部分：
    <?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE web-app
     PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
     "http://java.sun.com/dtd/web-app_2_3.dtd">
    <web-app>
    <filter>
            <filter-name>SetCharacterEncoding</filter-name>
            <filter-class>myFilter.ChangeCharsetFilter </filter-class>
            <init-param>
                <param-name>encoding</param-name>
                <param-value>GB2212</param-value>//////指定编码为 GB2212
            </init-param>
         </filter>
        <filter-mapping>
            <filter-name>SetCharacterEncoding</filter-name>
            <url-pattern>/*</url-pattern>////////对于所有的 request 改变其编码
        </filter-mapping>
    </web-app>
```

23.3 Java 与数据库之间的乱码

大部分数据库都支持 Unicode 编码方式，所以解决 Java 与数据库之间的乱码问题比较明智的方式是直接使用 Unicode 编码与数据库进行交互。很多数据库驱动自动支持 Unicode，如 Microsoft 的 SQL Server 驱动。其他大部分数据库驱动可以在驱动的 URL 参数中指定，例如，jdbc:oracle://localhost/WEBCLDB?useUnicode=true&character Encoding=GBK。如果是通过数据源连接，那么就要在配置数据源的配置文件进行设置，指定连接数据库是要使用的编码方式。

有时的乱码源自数据库自身。

例如，用 SQL Plus WorkSheet SQL 语句编辑器时，安装成功后，运行 SQL Plus WorkSheet 程序，出现所有的中文显示以及查询结果均为乱码的情况。查询安装时字符集设置 SQL→select userenv('language') from dual;，发现数据库安装时所选字符集为简体中文 ZHS16GBK，说明安装时字符集设置完全正确。第二步开始怀疑是用户客户端字符集问题，检查客户端注册表，打开注册表编辑程序（RegEdit），在 HKEY_LOCAL_MACHINE/SOFTWARE/ORACLE/ NLS_LANG 中发现 Oracle 客户端字符集为 AMERICAN_AMERICA.ZHS16

GBK，设置也完全正确，可以排除是客户端字符集设置错误的问题。

最后发现是 Oracle enterprise manage 中有一个配置文件名为 dbappscfg.properties，修改该文件即可解决上述问题。去掉注释符#，同时将其修改为 SQLPLUS_NLS_LANG=AMERICAN_AMERICA.ZHS16GBK。

对于 Windows 操作系统，还需要修改一项，在文件中找到# SQLPLUS_SYSTEMROOT=c:\\WINNT40，去掉注释符，将其修改为您所在机器的操作系统主目录。如操作系统的主目录在 d 盘的 Winnt 下，则将其修改为 SQLPLUS_SYSTEMROOT=d:\\WINNT。对于后面一项的修改只对 Windows 操作系统进行，对 UNIX 操作系统则不需要。

在安装完 Oracle10g 数据库后，em、bdca、netca 有时显示中文时出现乱码，也有可能是 jre 默认使用的字体导致的结果。可以查看$oracle_home/jdk 目录下 jre 的当前字体，不是*zh_CN*要改为*zh_CN*再删除 gif 下的默认文件，最后重启 EM 服务。

23.4　Java 与文件/流之间的乱码

Java 读写文件最常用的类是 FileInputStream/FileOutputStream 和 FileReader/FileWriter。其中 FileInputStream 和 FileOutputStream 是基于字节流的，常用于读写二进制文件。读写字符文件建议使用基于字符的 FileReader 和 FileWriter，省去了字节与字符之间的转换。

但这两个类的构造函数默认使用系统的编码方式，如果文件内容与系统编码方式不一致，可能会出现乱码。

在这种情况下，建议使用 FileReader 和 FileWriter 的父类：InputStreamReader/OutputStreamWriter，它们也是基于字符的，但在构造函数中可以指定编码类型：InputStreamReader(InputStream in, Charset cs) 和 OutputStreamWriter(OutputStream out, Charset cs)。

23.5　其他

运用上面的方法后，如果在其他地方还出现乱码，可能需要进行手动转码，通过调用类方法进行编码方式的转换。

示例 23-2：

```
package dbJavaBean;
public class CodingConvert
{
public CodingConvert()
{
        //process
}
public String toGb(String uniStr)
```

```
{
    String gbStr = "";
    if(uniStr == null)
      {
            uniStr = "";
    }
    try
      {
  byte[] tempByte = uniStr.getBytes("ISO8859_1");
  gbStr = new String(tempByte,"GB2212");
    }
     catch(Exception ex)
     {
            // exception process
    }
    return gbStr;
 }
 public String toUni(String gbStr)
 {
    String uniStr = "";
    if(gbStr == null)
      {
           gbStr = "";
    }
    try
      {
  byte[] tempByte = gbStr.getBytes("GB2212");
  uniStr = new String(tempByte,"ISO8859_1");
    }
      catch(Exception ex)
     {
    }
    return uniStr;
 }
}
```

也可以将获取的字符串直接用 ISO-8859-1 进行编码，然后将这个编码存放到一个字节数组中，然后将这个数组转化成字符串对象就可以了。

示例 23-3：

```
String str=request.getParameter("name");
Byte B[]=str.getBytes("ISO-8859-1");
Str=new String(B);
```

解决 Java 乱码问题的关键在于在字节与字符的转换过程中，转换时采用的编码必须前后一致。

补充：

由于操作系统、浏览器、数据库、JVM 采用的字符集都不一样，基于 WebLogic Server 开发的应用经常出现中文显示乱码问题，WebLogic Server 上运行的 Web 应用有很多与字

符集有关的设置，为了正确处理中文，最好把这些设置都设上。

（1）在 JSP 文件头加入：

```
<%@ page contentType="text/html; charset=GBK" %>
```

指定该 JSP 采用的字符集。

（2）在 weblogic.xml 文件的<jsp-descriptor>中加入：

```
<jsp-param>
<param-name>encoding</param-name>
<param-value>GBK</param-value>
</jsp-param>
```

指定 JSP 文件中采用的字符集，在 JSP 文件中的<%@ page contentType="text/html; charset=GBK" %>会覆盖该设置

（3）在 weblogic.xml 文件的<jsp-descriptor>中加入

```
<jsp-param>
<param-name>compilerSupportsEncoding</param-name>
<param-value>true</param-value>
</jsp-param>
```

如果为 true，指定在编译 JSP 文件时，采用在 JSP 文件中定义的<%@ page contentType="text/html; charset=GBK" %>或<jsp-descriptor>中定义的 encoding 字符集进行编码，如果为 false，则采用 JVM 中默认指定的字符集进行编码。

（4）WebLogic Server 需要把 HTTP request（GET 和 POST）中的数据从它的原始编码转化为 Unicode，以便 Java servlet API 进行处理，为了做这种转换，WebLogic Server 需要知道 HTTP request 中的数据的编码方式。这可以通过在 weblogic.xml 的<context-param>中设置。

示例 23-4：

```
<input-charset>
<resource-path>/</resource-path>
<java-charset-name>GBK</java-charset-name>
</input-charset>
```

或是在项目的 web.xml 添加如下内容：

示例 23-5：

```
<web-app>
    <context-param>
   <param-name>weblogic.httpd.inputCharset./*</param-name>
   <param-value>GB2212</param-value>
   </context-param>
   <context-param>
   <param-name>weblogic.httpd.inputCharset./*</param-name>
   <param-value>GBK</param-value>
```

```
    </context-param>
  </web-app>
```

（5）从 Oracle 数据库中检索出来的中文显示不正确时，在这种情况下，如果数据库使用的是中文字符集，并使用的是 Type 2 JDBC Driver 时，可加入 weblogic.codeset=GBK 的属性来解决这个问题，代码如下。

示例 23-6：

```
java.util.Properties props = new java.util.Properties();
props.put("weblogic.codeset", "GBK");
props.put("user", "scott");
props.put("password", "tiger");
String connectUrl = "jdbc:weblogic:oracle";
Driver myDriver = (Driver)
Class.forName("weblogic.jdbc.oci.Driver").newInstance();
Connection conn =myDriver.connect(connectUrl, props);
```

如果是采用 WTC 调用 Tuxedo 中的服务，在 JSP 页面中无法正确显示中文，必须使安装 Tuxedo 的服务器上的 NLS_LANG 环境变量与数据库中的字符集的设置一样。如果后台 Oracle 数据库中的字符集设置为 SIMPLIFIED CHINESE_CHINA.ZHS16GBK，那么 Tuxedo 应用服务器上的 NLS_LANG 环境变量应设置为 export NLS_LANG="SIMPLIFIED CHINESE_CHINA.ZHS16GBK"。

关于 AIX(Linux)WebLogic 中文乱码问题如下。

在启动文件 startWebLogic.sh 中加入 export LANG=zh_CN（在 Linux 下，可能是 export LANG=zh_CN.UTF-8）。

示例 23-7：

```
JAVA VENDOR="BEA"

JAVA_HOME="/bea/jrockit81sp6_142_10"
export LANG=zh_CN
```

23.6 关于 WebLogic 的国际化

在 WebLogic Server 中，可以设置范围不同的编码。例如，JSP 有一个用来指定页面特定编码的 page 标记，该 page 标记符合 JSP 1.2 规范。在使用 WebLogic jDriver 时，还可以使用 weblogic.codeset 属性来为 JDBC 连接指定编码。注意：为特定范围指定的编码与 WebLogic Server 所在的 Java VM 的默认编码没有任何关系。即便 Java VM 以英语区域设置运行，使用简体中文 JSP 文件提供服务也不会出现问题。但是，对下列字符串的处理取决于 Java VM 的默认编码。

- WebLogic Server 的错误日志输出。
- 与本地文件系统之间的文件输入和输出。

这些字符串使用平台的 Java VM 默认编码（由 -Dfile.encoding Java 系统属性指定的

编码）。例如，WebLogic Server 输出到终端控制台的日志消息的语言和编码取决于在 Java VM 中指定的编码。如果希望切换 WebLogic Server 日志消息的语言和编码，则需要对系统区域设置进行相应切换。一旦 VM 启动，就不能动态切换 Java VM 默认编码。在重新启动 WebLogic Server 之前，须确保进行了下列设置。

Windows 2000 或 Windows NT 系统中，在“控制面板”页的“区域选项”栏中选择“英语（美国）”或“简体中文”选项。这允许服务器使用 CP1242 或 MS936 作为默认编码。

UNIX 系统中，在 LANG 环境变量中指定您的平台支持的区域设置。

表 23-1 列出了服务器编码和 LANG 环境变量的设置。

表 23-1

平台	编码	LANG 环境变量
Solaris	GB2212、GBK、GB18030	zh_CN、zh_CN.GBK 或 zh_CN.GB18030
HP	GB2212	zh_CN.hp15CN

例如，如果为 Solaris 指定 GB2212，则 LANG 的设置应如下所示：

```
LANG=zh_CN
```

如何检查服务器编码？

Java VM 默认编码变成 WebLogic Server 默认编码。可以通过参考管理控制台中的日志消息来检查编码。检查步骤如下。

（1）在管理控制台上，用鼠标在左窗格中右击服务器名，然后在弹出的快捷菜单中执行“查看服务器日志”命令。

（2）单击“自定义此视图”按钮。

（3）在“子字符串”文本框中，输入“file.encoding”。

（4）单击“应用”按钮。

所显示的编码就是服务器编码。

在 WebLogic Server 中，对于一个域中的所有服务器必须使用相同的编码，这意味着在使用 WebLogic Server 时，应当用相同的编码配置该集群中的所有服务器。

例如，如果集群中有一个 Windows 平台，则该集群中所有的编码都应当是相同的 GB2212 类型的编码，如 MS936。如果某个服务器的编码不同于存在的其他编码，则该服务器的日志可能无法正确显示。

config.xml 文件是 UTF-8 形式的输入/输出。在用文本编辑器直接编辑该文件时，可以 UTF-8 形式进行读取和保存。

关于将 WebLogic Server 用作 Web 服务器的说明如下。

要将 contentType 字符集参数添加到 HTTP 头中以提供 HTML 文件，可在 web.xml 文件中插入下列定义，这些定义可以为 HTML 文件显式指定编码。

示例 23-8：

```
<mime-mapping>
  <extension>html</extension>
```

```
    <mime-type>text/html;charset=GB2212</mime-type>
  </mime-mapping>
```

这允许您通过使用如下所示的 META 标记来忽略 HTML 文件中的字符集设置：

```
<META HTTP-EQUIV="content-type" CONTENT="text/html; charset=GB2212">
```

在创建 JDBC 连接池时，对于使用多字节字符的数据库，必须为到该数据库的连接指定适当的编码。您可能需要对 Web 层和 DB 层之间的编码转换映射进行匹配。有关详细信息，可参阅 WebLogic 在线文档 Configuring and Using WebLogic jDriver for Oracle 中 Advanced Oracle Features 一章的 Codeset Support。

在 WebLogic Server 的 J2EE 组件的 DD 文件中，多字节字符的编码按照 XML 声明中指定的编码进行处理。如果 DD 文件中没有 XML 声明或者 XML 声明中没有编码特性，将以 UTF-8 编码处理该文件。

在 WebLogic Builder 或管理控制台中编辑 DD 文件并保存所做的更改时，该文件的编码将与初始文件中的相同。WebLogic Builder 中新建的 DD 文件可能没有 XML 声明。更改这些文件的编码时，可在 XML 声明中设置编码特性，并为该文件设置相应的编码转换。

管理控制台首次启动时显示的语言是在 Web 浏览器中指定的首选语言。例如，如果使用的是简体中文版本的 Windows 和 Internet Explorer，则管理控制台在首次启动时将显示简体中文。如果希望将首次显示的语言更改为英语，可以将浏览器中的首选语言设置为英语。

在 WebLogic Server 8.1 中可以选择的管理控制台语言如下。

（1）简体中文/EUC-CN。

（2）简体中文/GBK。

（3）简体中文/GB18030。

（4）英语。

根据管理控制台所连接到的管理服务器的编码选择 GB2212、GBK 或 GB18030。在管理控制台启动之后要想切换语言，可以在管理控制台主页的“首选项”页上，从“语言”下拉列表框中选择所需的语言。

23.7 关于 WebLogic 的日志乱码

（1）对于 Windows，在如下目录（由安装路径和域名确定）中找到 setDomainEnv.cmd：

```
..\bea\user_projects\domains\mydomain\bin
```

用文本编辑器对 setDomainEnv.cmd 进行修改。

找到 set JAVA_OPTIONS=%JAVA_OPTIONS%后添加-Dfile.encoding=utf-8（如图 23-1 所示）。

（2）对于 Linux（乱码 SSH 日志）：

```
vi /etc/sysconfig/i18n
```

```
set JAVA_OPTIONS=%JAVA_OPTIONS% -Dfile.encoding=utf-8

@REM SET THE CLASSPATH
```

图 23-1

将内容改为：

```
LANG="zh_CN.GB18030"
LANGUAGE="zh CN.GB18030:zh CN.GB2212:zh CN"
SUPPORTED="zh_CN.GB18030:zh_CN:zh:en_US.UTF-8:en_US:en"
SYSFONT="lat0-sun16"
```

这样中文在 SSH、telnet 终端就可以正常显示了。

第 24 章　WebLogic 集群故障

24.1　问题定位

24.1.1　集群概述

集群是命名的服务器实例集合，它们共享相同的应用程序、资源和配置信息，您可以将不同计算机上的服务器实例分组到一个逻辑集群中并将其作为一个单元来管理。

集群可以启用水平可伸缩性、负载平衡和故障转移保护。根据定义，集群中的所有实例都具有相同的资源和应用程序配置。当集群中的服务器实例或计算机出现故障时，负载平衡检测到该故障，会将通信从出现故障的实例重定向至集群中的其他实例，并恢复用户会话状态。由于集群中所有实例上的应用程序和资源都相同，因此一个实例可以故障转移至集群中的任何其他实例中。

24.1.2　如何检测集群故障

（1）集群中的 WebLogic Server 实例通过监视下列内容来检测其对等服务器实例的失败。

① 与对等服务器的 Socket 连接。

② 定期的服务器心跳消息。

（2）使用 IP Socket 的失败检测。

WebLogic Server 实例将监视对等服务器实例之间的 IP Socket 使用情况作为检测失败的即时方法。如果某个服务器连接到了集群中的一个对等服务器上，并开始通过 Socket 传输数据，则该 Socket 的意外关闭会导致该对等服务器被标记为“失败”，其关联服务也会从 JNDI 命名树中删除。

（3）WebLogic Server “心跳”

如果集群的服务器实例对于对等通信没有打开的 Socket，则还可以通过 WebLogic Server 心跳检测失败的服务器。集群中的所有服务器实例都使用多播向集群的其他成员广播定期的服务器心跳消息。每个心跳消息都包含唯一标识发送该消息的服务器的数据。服务器以 10 秒的定期间隔广播其心跳消息。而集群中的每个服务器又会监视多播地址以确保所有对等服务器的心跳消息都在进行发送。

如果监视多播地址的服务器丢失了来自对等服务器的 3 个心跳（即如果它在 30 秒或更长的时间内没有收到来自该服务器的心跳），则监视服务器会将该对等服务器标记为“失败”。然后它会根据需要更新其本地 JNDI 树，以取消承载在该失败服务器上的服务。

因此，即使服务器没有为对等通信打开 Socket，它们也可以检测失败。

24.2　集群常规配置

24.2.1　一般性配置

1．名称

集群配置的名称。WebLogic Server 使用 MBean 实现和持久保存配置。

注：更改将在重新部署模块或重新启动服务器后生效。

2．集群地址

作为客户端连接到此集群所使用的 URL 的一部分的地址，用于生成 EJB 句柄和实体 EJB 故障转移地址。地址信息可以是 IP 地址或 DNS 名称和端口号。决定使用 DNS 名称还是 IP 地址时，要考虑集群的用途。用于生产环境时，通常建议使用 DNS 名称。在下列情况下使用 IP 地址可能会造成转换错误。

客户端要穿越防火墙连接集群。在展示层和对象层之间具有防火墙，例如，在 Servlet 集群和 EJB 集群之间具有防火墙。

① 通过将单个服务器实例的地址绑定到 DNS 名称可以避免转换问题。确保环境中防火墙两侧的服务器实例的 DNS 名称相同，不要使用同时也是网络上某个 NT 系统名称的 DNS 名称。

② 如果配置了网络通道，则可以为每个通道设置集群地址。

注：更改将在重新部署模块或重新启动服务器后生效。

3．默认负载算法

负载算法有 round-robin、weight-based、random、round-robin-affinity、weight-based-affinity、random-affinity。默认为循环法算法（round-robin）。该选择为没有为特定服务指定任何算法时复制的服务之间的负载平衡将使用的算法。循环法算法按顺序循环完成 WebLogic Sever 的实例列表。基于权重的负载平衡在循环法算法基础上做了改进，它将每个服务器的预分配权重都考虑在内。在随机负载平衡中，以随机方式发送请求到服务器。

4．已启用 WebLogic 插件

如果集群将从代理插件或 HttpClusterServlet 中接受请求，则将此特性设置为 true。如果 WebLogicPluginEnabled 为 true，则对 getRemoteAddr 的调用将从专用 WL-Proxy-Client-IP 头（而不是 Web 服务器）中返回浏览器客户端地址。

注：更改将在重新部署模块或重新启动服务器后生效。

5．服务期限阈值

该值介于 0 到 65534 之间时是合法的。在确定一个服务早于另一个服务时，这两个发

生冲突的服务之间的期限差必须达到的数秒。

6. 已启用客户端证书代理

如果针对某个集群设置为 true，则此特性将指定，此集群的服务器上承载的 Web 应用程序客户端的证书将包括在由代理插件或 HttpClusterServlet 发送的特殊 WL-Proxy-Client-Cert 头中。（可以在 web.xml 中的集群、服务器和 Web 应用程序级别定义 ClientCertProxyEnabled。）

在代理服务器上执行用户身份验证时，此设置非常有用。如果将 clientCertProxy 设置为 true，则会使代理服务器将证书传递给特殊头 WL-Proxy-Client-Cert 中的集群。

WL-Proxy-Client-Cert 头可以由任何可以访问 WebLogic Server 的客户端提供。WebLogic Server 从此头中获得证书信息，信任其来自于安全来源（插件）并使用此信息对用户进行身份验证。

为此，如果将 clientCertProxy 设置为 true，请使用连接筛选器确保 WebLogic Server 仅接受运行此插件的计算机上的连接。

注：更改将在重新部署模块或重新启动服务器后生效。

24.2.2 Multicast 相关配置

注：以下设置将在重启服务器后生效。

1. 多播地址

多播地址应为介于 214.0.0.0 到 229.245.245.245 之间的 IP 地址，默认为 229.192.0.0；不可以设置为*.0.0.1 类的值。

2. 多播端口

该值介于 1 到 65535 之间时合法，用于集群的成员之间的互相交流。

3. 多播消息发送延迟

该值介于 0ms 与 100ms 之间时合法。为了避免缓冲溢出，可以延迟发送消息。

4. 多播生存时间

该值介于 1～245 之间时合法。如果您的集群跨越 WAN 中的多个子网，则该集群的多播生存时间参数值必须足够高，可以确保路由器不会再多播数据包到达其最终目标之前放弃这些多播数据包。多播 TTL 参数设置可以放弃数据包之前多播消息进行的网络跃点数。对多播 TTL 参数进行合适配置会降低在集群服务器实例之间传输的多播消息发生丢失的危险。要为集群配置多播 TTL，需要在管理控制台用于该集群的“多播”选项卡中更改“多播 TTL”值。

下面为设置 TTL 的 config.xml 文件显示。

示例 24-1：

```
<Cluster
    Name="testcluster"
    ClusterAddress="wanclust"
    MulticastAddress="wanclust-multi"
    MulticastTTL="3"
/>
```

其多播生存时间为 3，即确保集群的多播消息在放弃之前可以穿越 3 个路由器。

5．多播缓冲区大小

如果因为集群中的服务器实例未及时处理传入的消息而发生多播风暴，则可以增加多播缓冲区的大小，最小为 64KB。

24.3　集群负载均衡

24.3.1　负载均衡两方面的定义

（1）把大量的并发访问或数据流量分担到多台节点设备上分别处理，减少用户等待响应的时间。

（2）单个重负载的运算分担到多台节点设备上并行处理，每个节点设备处理结束后，将结果汇总，再返回给用户，使得信息系统处理能力可以得到大幅度提升。

24.3.2　Servlet 和 JSP 的负载平衡

Servlet 和 JSP 的负载平衡可以通过 WebLogic 代理插件的内置负载平衡功能或者单独的负载平衡硬件实现。

1．使用代理插件进行负载平衡

WebLogic 代理插件维护一个 WebLogic Server 实例列表（这些实例承载着集群 Servlet 或 JSP），并以循环法为基础向这些实例转发 HTTP 请求。这种负载平衡方法在循环法负载平衡中进行了描述。

该插件还提供在某个 WebLogic Server 实例失败时查找客户端 HTTP 会话状态副本所需的逻辑。

WebLogic Server 支持下列 Web 服务器和相关联的代理插件。

（1）带有 HttpClusterServlet 的 WebLogic Server。

（2）带有 Netscape（代理）插件的 Netscape Enterprise Server。

（3）带有 Apache Server（代理）插件的 Apache。

（4）带有 Microsoft-IIS（代理）插件的 Microsoft Internet Information Server。

2．使用外部负载平衡器实现 HTTP 会话的负载平衡

使用外部负载平衡解决方案的集群可以使用该硬件支持的任何负载平衡算法。其中可能包括监视每个计算机使用情况的基于负载的高级平衡策略。

（1）负载平衡器配置要求

如果选择使用负载平衡硬件而不是代理插件，则该硬件必须支持兼容的被动或主动 Cookie 持久性机制和 SSL 持久性。

① 被动 Cookie 持久性。

被动 Cookie 持久性使得 WebLogic Server 能够将包含会话参数信息的 Cookie 通过负载平衡器写入客户端。

② 主动 Cookie 持久性。

某些主动 Cookie 持久性机制可用于 WebLogic Server 集群，前提是负载平衡器不修改 WebLogic Server Cookie。WebLogic Server 集群不支持覆盖或修改 WebLogic HTTP 会话 Cookie 的主动 Cookie 持久性机制。如果负载平衡器的主动 Cookie 持久性机制通过向客户端会话添加其自己 Cookie 的方式运行，则将该负载平衡器用于 WebLogic Server 集群时不需要任何附加配置。

③ SSL 持久性。

使用 SSL 持久性时，负载平衡器执行客户端和 WebLogic Server 集群之间数据的所有加密和解密操作。然后负载平衡器使用 WebLogic Server 插入客户端的纯文本 Cookie 来维护客户端和集群中特定服务器之间的关联。

（2）负载平衡器和 WebLogic 会话 Cookie

使用被动 Cookie 持久性的负载平衡器可以使用 WebLogic 会话 Cookie 中的字符串将客户端与承载其主 HTTP 会话状态的服务器相关联。该字符串可唯一标识集群中的服务器实例。您必须使用该字符串常量的偏移和长度配置负载平衡器。偏移和长度的正确值取决于会话 Cookie 的格式。会话 Cookie 的格式为：

```
sessionid!primary_server_id!secondary_server_id
```

其中：sessionid 为 HTTP 会话随机生成的标识符，该值的长度由应用程序 weblogic.xml 文件<session-descriptor>元素中的 IDLength 参数配置。默认情况下，sessionid 长度为 52 个字节。

Primary_server_id 与 secondary_server_id 为会话主要主机和次级主机的 10 字符标识符。

24.3.3 EJB 和 RMI 对象的负载平衡

对象的负载平衡算法在为集群对象获得的副本感知存根控件中得到维护。

默认情况下，WebLogic Server 集群使用循环法负载平衡。您可以通过使用管理控制台来设置 weblogic.cluster.defaultLoadAlgorithm，从而为集群配置不同的默认负载平衡方法。

您还可以使用 rmic 中的-loadAlgorithm 选项，或使用 EJB 部署描述符中的 home-load-algorithm 或 stateless-bean-load-algorithm，为特定 RMI 对象指定负载平衡算法。为对象配置的负载平衡算法会替换集群的默认负载平衡算法。

除了标准负载平衡算法之外，WebLogic Server 还支持自定义的基于参数的路由。

24.3.4　JMS 的负载平衡

默认情况下，WebLogic Server 集群使用循环法对对象进行负载平衡。要使用 JMS 对象提供服务器关系的负载平衡方法，必须将集群作为整体为其配置所需的方法。您可以通过使用管理控制台设置 weblogic.cluster.defaultLoadAlgorithm 来配置负载平衡算法。

1. 分布式 JMS 目标的服务器关系

对于使用分布式目标功能的 JMS 应用程序，将支持服务器关系；此功能对于独立目标则不受支持。如果为 JMS 连接工厂配置服务器关系，则在分布式目标的多个成员间对使用者或生产者进行负载平衡的服务器实例将首先尝试在同时运行于同一个服务器实例的任何目标成员之间进行负载平衡。

2. 客户端连接的初始上下文关系和服务器关系

系统管理员可以配置多个 JMS 服务器并使用目标将它们分配到定义的 WebLogic Server，从而在集群中的多个服务器之间建立 JMS 目标的负载平衡。每个 JMS 服务器仅部署于一个 WebLogic Server 上，并处理一组目标的请求。在配置阶段中，系统管理员通过为 JMS 服务器指定目标来启用负载平衡。

24.3.5　JDBC 连接的负载平衡

JDBC 连接的负载平衡需要使用为负载平衡配置的多数据源。负载平衡支持是配置多数据源时可以选择的选项。

负载平衡多数据源提供了高可用行为，并在该多数据源中的多个数据源之间平衡负载。多数据源具有它所包含的数据源的顺序列表。如果未配置用于负载平衡的多数据源，则它总是尝试从列表中的第一个数据源获取连接。在负载平衡多数据源中，将使用循环法方案访问它所包含的数据源。在多数据源连接的每个后续客户端请求中，该列表都会旋转，以便第一个共享池会接触该列表中的循环。

24.3.6　负载平衡器的算法

负载平衡器使用不同的算法控制通信流量。这些算法用于以智能方式分散负载，并/或最大限度地利用集群内的所有服务器。这些算法中的一部分示例包括下列内容。

1. 循环法负载平衡

循环算法将负载均衡地分配给每台服务器，而不考虑当前的连接数或响应时间。循环

法适合于集群中的服务器具有相同处理能力的情况；否则，一些服务器收到的请求可能会超过它们的处理能力，而其他服务器的处理能力则有富余。WebLogic Server 使用循环法算法作为没有指定算法时集群对象存根控件的默认负载平衡策略。对 RMI 对象和 EJB 来说，均支持此算法。它还是 WebLogic 代理插件使用的方法。

循环法算法的优点在于它简单、使用资源少而且极具有预见性，主要缺点在于可能会发生护航。当一个服务器的速度明显慢于其他服务器时会发生护航。因为副本感知存根控件或代理插件以同一顺序访问服务器，所以速度较慢的服务器可以导致针对该服务器请求同步，然后将该服务器排在其他服务器的后面以应对未来的请求。

2．基于权重的负载平衡

此算法仅适用于 EJB 和 RMI 对象集群。

基于权重的负载平衡通过考虑为每个服务器预先分配的权重，在循环法基础上进行了改善。您可以使用管理控制台中的“服务器”→“配置”→“集群”选项卡，在“集群权重”字段中为集群中的每个服务器分配一个介于 1 和 100 之间的数字权重。此值决定了服务器相对于其他服务器要承担的负载比例。如果所有服务器的权重相同，它们每个则会承担相同比例的负载。如果一个服务器的权重为 50，而其他所有服务器的权重都是 100，则这个权重为 50 的服务器承担的负载为所有其他服务器的一半。

要分配给每个服务器实例的相对权重要考虑以下两个因素。

（1）服务器硬件处理能力与其他服务器之间的相对比例（例如专用于 WebLogic Server 的 CPU 的数量和性能）。

（2）每个服务器承载的非集群（固定）对象的数量。

注：对于使用 RMI/IIOP 协议进行通信的对象，不支持基于权重的负载平衡。

3．随机负载平衡

随机负载平衡方法仅适用于 EJB 和 RMI 对象集群。

在随机负载平衡中，请求是随机路由到服务器的。建议只对于每个服务器实例在配置相似的计算机上运行均匀集群部署时才使用随机负载平衡。请求的随机分配不允许服务器实例运行所在的计算机之间存在处理能力差异。如果集群中承载服务器的某个计算机的处理能力比集群内的其他计算机差很多，随机负载平衡仍然会为处理能力较差的计算机提供与处理能力较强的计算机同样多的请求。

随机平衡负载在集群的服务器实例之间均匀分布请求，并且会随着请求累积数量的增加而增加。对于较少数量的请求，该负载可能无法精确均匀平衡。

随机负载平衡的缺点包括生成每个请求的随机数量所产生的微小处理开销，以及对于较小数量的请求负载可能无法均匀平衡。

4．服务器关系负载平衡算法

WebLogic Server 为提供服务器关系的 RMI 对象提供了 3 种负载平衡算法：循环法关系、基于权重关系、随机关系。

对于多有类型的 RMI 对象（包括 JMS 对象）、多有 EJB Home 接口和无状态 EJB 远程接口均支持服务器关系。

服务器关系对于外部客户端连接会关闭负载平衡，而客户端在选择要在其中访问对象

的服务器实例时会考虑其与 WebLogic 服务器实例的现有连接。如果某个对象针对服务器关系进行了配置，客户端存根控件则会尝试选择它已经连接的服务器实例，并继续将同一服务器实例用于方法调用。该客户端上的存根控件都会尝试使用该服务器实例。如果该服务器实例不可用，存根控件则会在可能的情况下故障转移到客户端已经连接的服务器实例中。

服务器的目的是为了尽可能减少外部 Java 客户端和集群中的服务器实例之间打开的 IP 套接口数。

WebLogic Server 通过使得针对对象的方法调用“粘连”到现有连接的方式实现上述目的，而不在可用服务器实例之间对这些方法调用进行负载平衡。使用服务器关系算法时，使用资源较少的服务器到服务器的连接仍然根据配置的负载平衡算法进行负载平衡，仅对于外部客户端连接才禁用负载平衡。

服务器关系算法在 WebLogic 服务器实例之间平衡客户端负载时会考虑外部 Java 客户端和服务器实例之间的现有连接。服务器关系如下。

（1）会关闭外部 Java 客户端和服务器实例之间的负载平衡。

（2）会使得来自外部 Java 客户端的方法调用与该客户端已经具有打开连接（假设该连接支持必要的协议和 QOS）的服务器实例相粘连。

（3）在发生失败时会使得客户端故障转移到它具有打开连接（假设该连接支持必要的协议和 QOS）的服务器实例。

（4）不影响为服务器到服务器连接执行的负载平衡。

5．示例 1——来自集群的上下文

在此示例中，客户端从集群获取上下文。在上下文中的查找以及对象调用粘连到一个连接。对新初始上下文的请求基于循环法进行负载平衡。

客户端从集群获取上下文如图 24-1 所示。

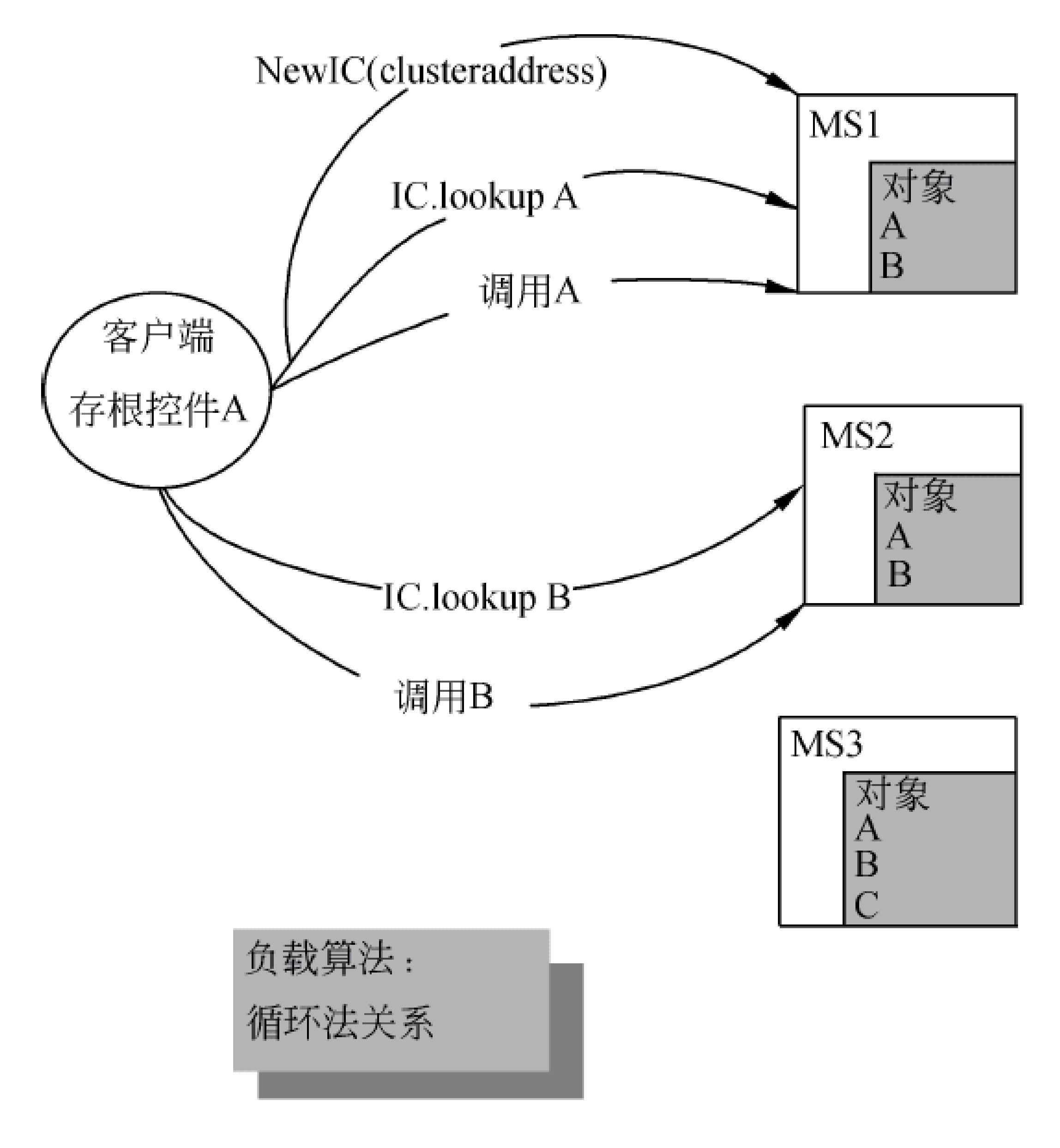


图 24-1

（1）客户端从集群请求新的初始上下文（Provider_URL=clusteraddress），并从 MS1 获取该上下文。

（2）客户端在对象 A 的上下文上执行查找，查找转至 MS1。

（3）客户端发出对对象 A 的调用。该调用转至该客户端已经连接的 MS1。对于对象 A 的附加方法调用粘连到 MS1。

（4）客户端从集群请求新的初始上下文（Provider_URL=clusteraddress），并从 MS2 获取该上下文。

（5）客户端在对象 B 的上下文上执行查找。该调用转至该客户端已经连接的 MS2。对于对象 B 的附加方法调用粘连到 MS2。

6．示例 2——服务器关系和故障转移

此示例演示了服务器关系对对象故障转移的影响。当受管服务器关闭时，客户端会故障转移到与它具有连接的另一个受管服务器上。

服务器关系和故障转移如图 24-2 所示。

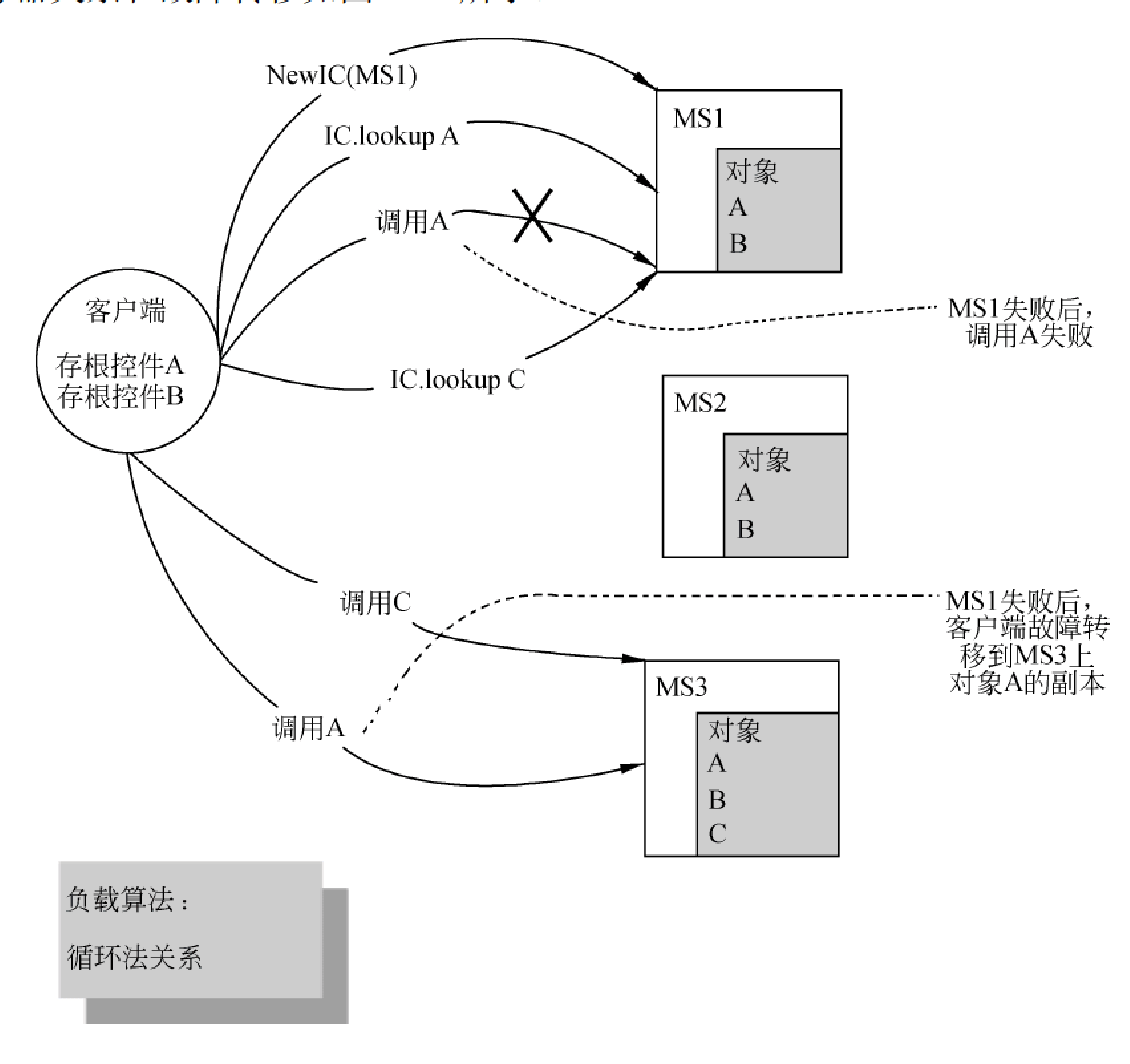


图 24-2

（1）客户端从 MS1 请求新的初始上下文。

（2）客户端在对象 A 的上下文中执行查找，查找转至 MS1。

（3）客户端对对象 A 进行调用。该调用转至该客户端已经连接的 MS1。对于对象 A 的附加调用粘连到 MS1。

（4）客户端获取对象 C 的存根控件，该对象是固定到 MS3 的。客户端打开到 MS3 的连接。

（5）MS1 失败。

（6）客户端对对象 A 进行调用。客户端不再具有到 MS1 的连接。因为客户端连接到了 MS3，所以它的故障转移到了 MS3 上对象 A 的副本。

7．示例 3——服务器关系和服务器到服务器连接

此示例说明了服务器关系对服务器实例之间的连接不会产生影响的事实。

服务器关系和服务器到服务器的连接如图 24-3 所示。

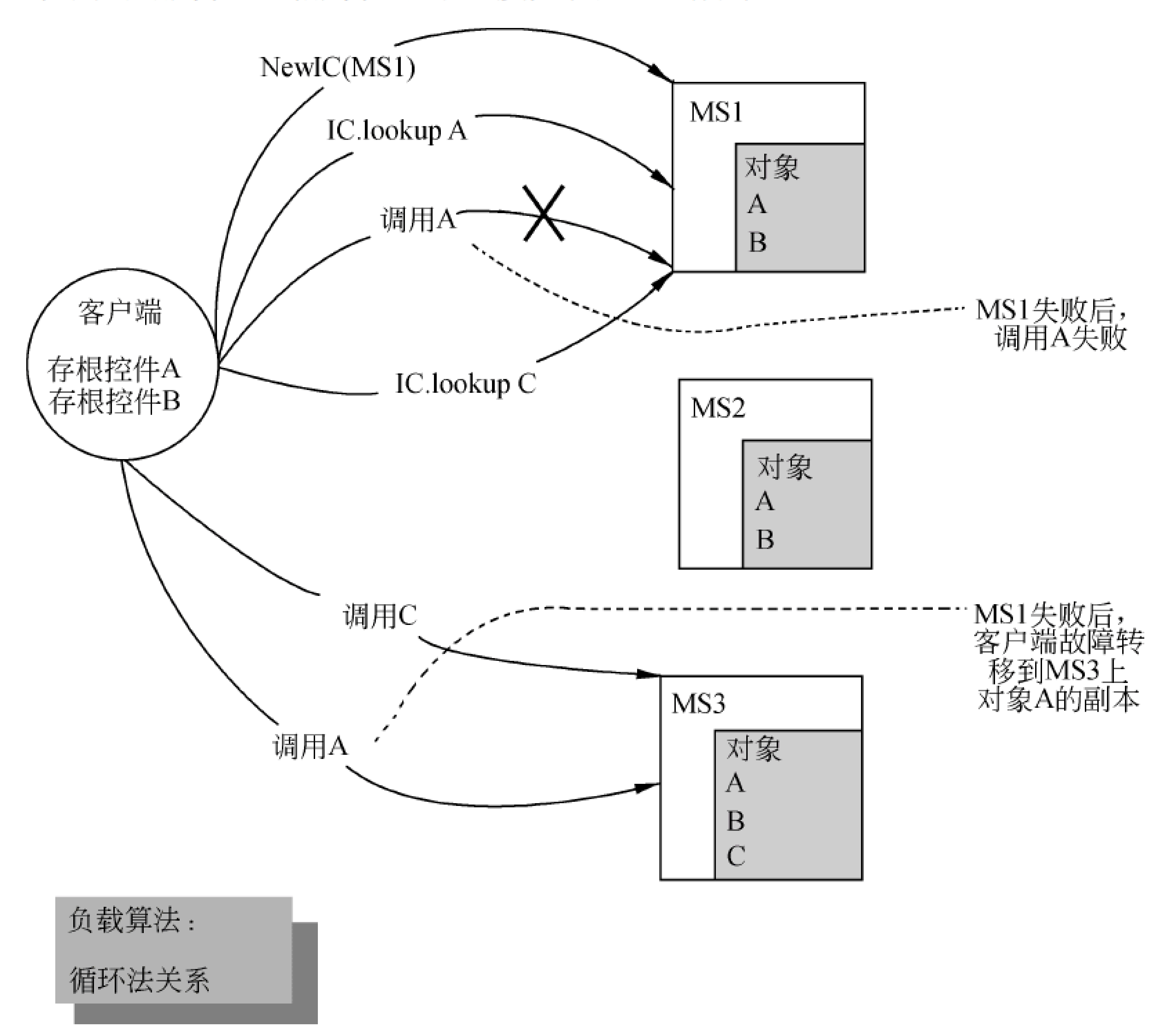


图 24-3

（1）MS4 上的 JSP 获取了对象 B 的存根控件。

（2）该 JSP 选择了 MS1 上的副本。对于每个方法调用，该 JSP 会以循环法为基础循环选择对象 B 在其中可用的受管服务器。

24.3.7　方法补充——共存对象的优化

WebLogic Server 并不总是对对象的方法调用进行负载平衡。在大多数情况下，使用与存根控件本身共存的副本效率更高，而不要使用位于远程服务器上的副本。

共存优化替换方法调用的负载平衡逻辑如图 24-4 所示。

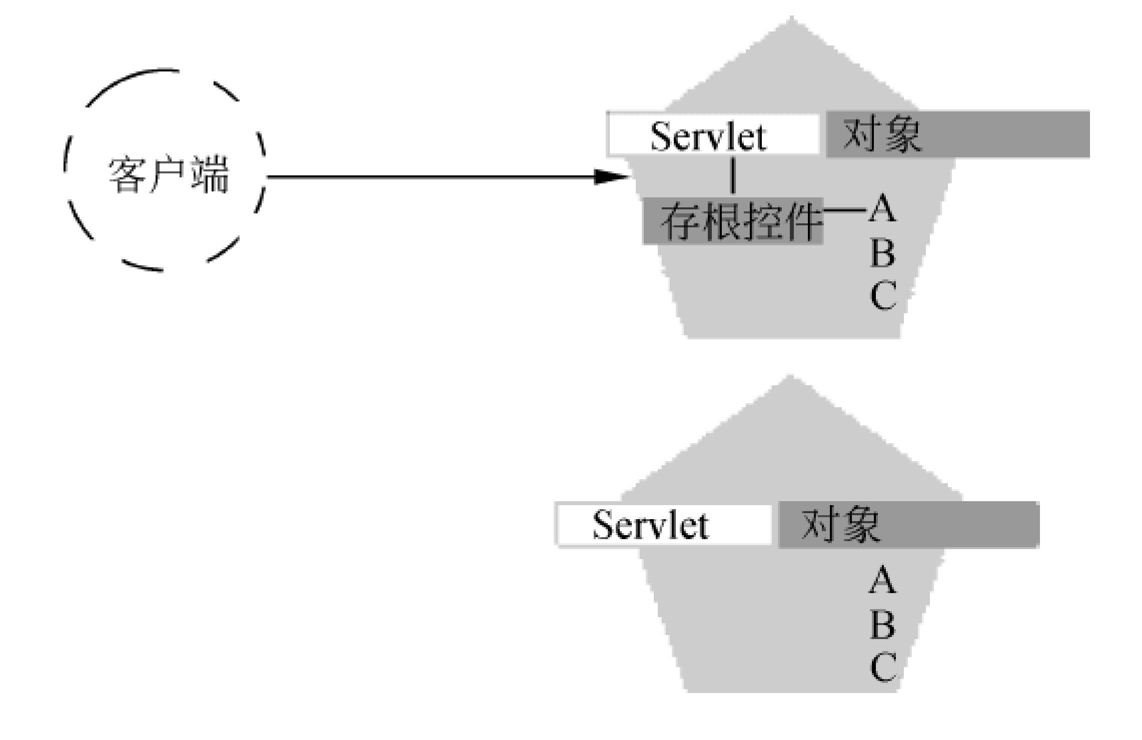


图 24-4

在此实例中，客户端连接集群中的第一个 WebLogic Server 实例承载 Servlet。为了响应客户端活动，该 Servlet 获取了对象 A 的副本感知存根控件。因为对象 A 的副本在同一服务器实例上也可用，所以我们说该对象与该客户端的存根控件是共存的。

WebLogic Server 总是使用对象 A 的本地共存副本，而不是将客户端的调用分到集群中对象 A 的其他副本中，使用本地副本的效率更高，因为使用本地副本可避免与集群的其他服务器之间建立对等连接的网络开销。

这种优化在计划 WebLogic Server 集群时经常被忽略。如果 Web 应用程序部署到一个集群中，共存优化则会替换副本感知存根控件中固有的任何负载平衡逻辑。

事务共存。作为基本共存策略的扩展，WebLogic Server 会尝试使用作为同一事务的一部分列出的共存集群对象。当客户端创建 UserTransaction 对象时，WebLogic Server 会尝试使用与该事务共存的对象副本。这种优化在图 24-5 中进行了描述。

共存优化扩展到事务中的其他对象图如图 24-5 所示。

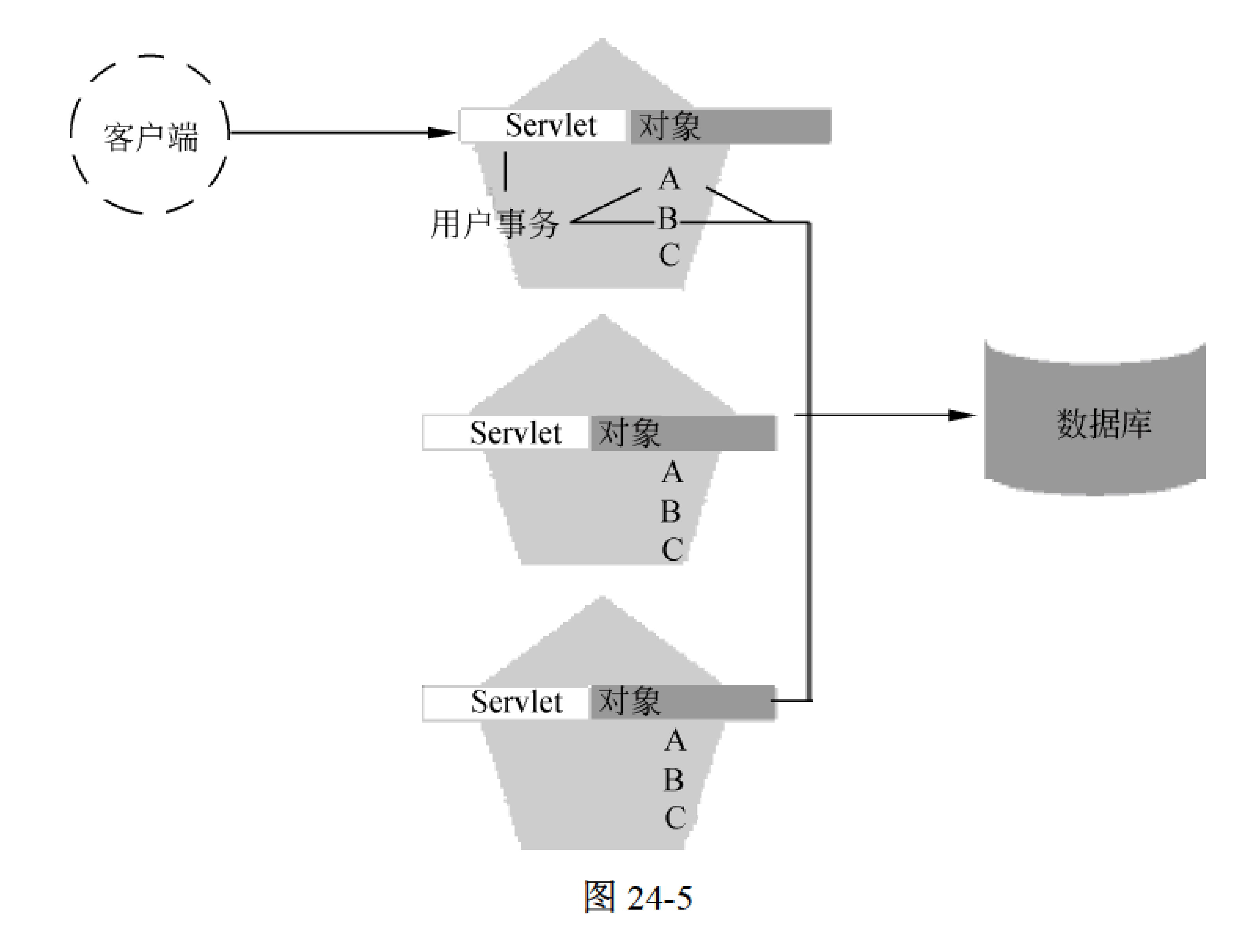


图 24-5

在此实例中，客户端连接集群中的第一个 WebLogic Server 实例，并获取 UserTransaction 对象。开始新事物之后，该客户端查找对象的 A 和 B 的副本，而不管 A 和 B 的存根控件

中负载平衡策略如何。

24.3.8 负载平衡器的故障检测功能

平衡器可以跟踪服务器或在服务器上运行的应用程序，并在服务器出现故障后停止向该服务器发送请求。

图 24-6 显示了负载平衡的基本组件。

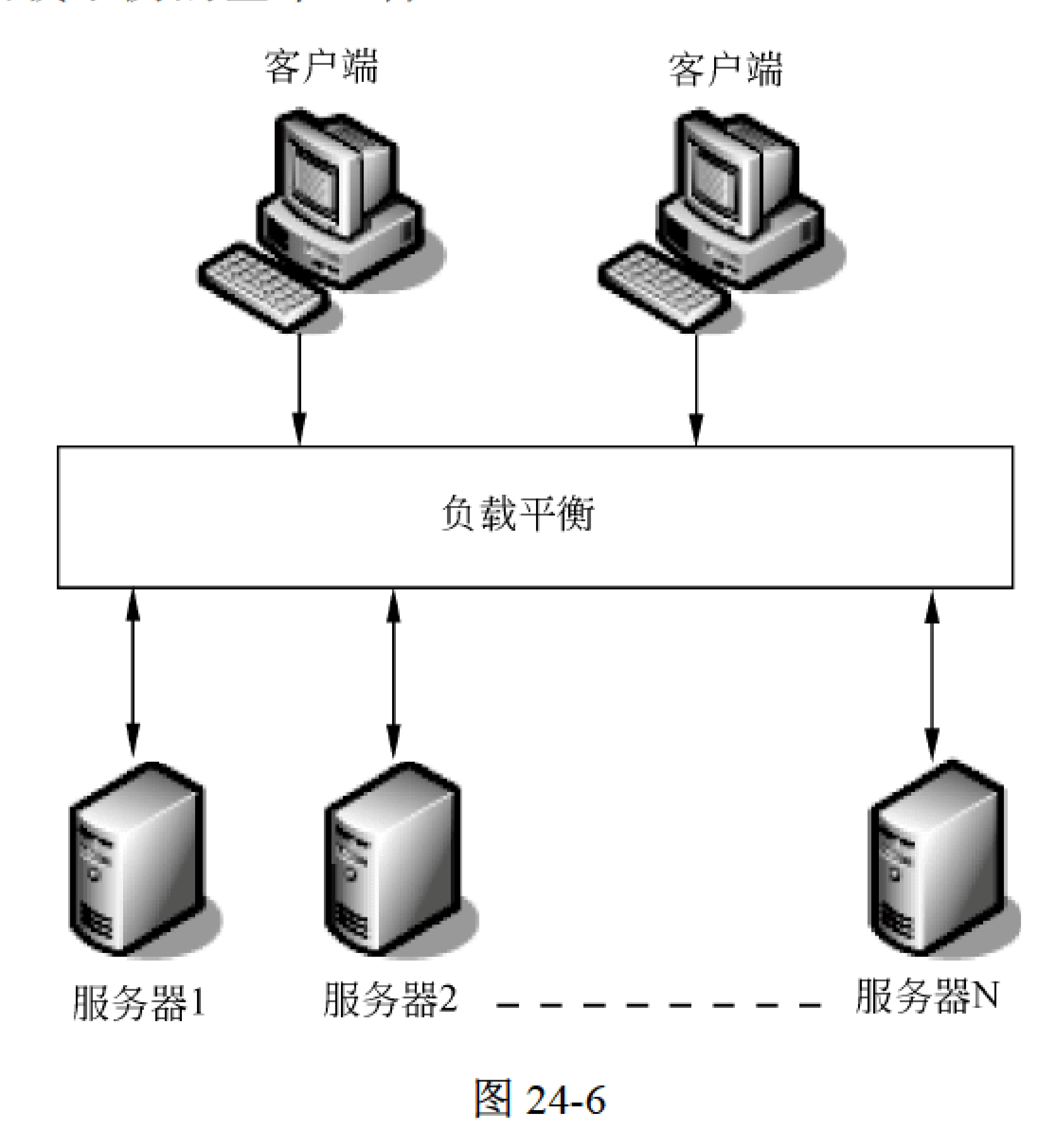


图 24-6

当负载平衡器收到来自客户端的请求时，集群组中的一台服务器将处理该请求。每台服务器都能够独立地处理请求。如果任何服务器因出现错误或正在维护而不可用，其他服务器仍然可以为请求提供服务而不会受到影响。因此，服务的总体可用性比由单台服务器处理所有请求的方案要高得多。但是，如果在一组软件负载平衡服务器前面使用单个物理负载平衡器或单个网络交换机，将会引入另一个故障单点。可以使用冗余负载平衡设备和/或交换机来减少这类风险。

24.3.9 非负载平衡层与负载平衡层的对比

1. 非负载平衡层

采用类似于图 24-7 中所描述的解决方案体系结构，该体系结构可能满足最初的性能要求。但是，随着负载的增加，应用程序层必须适应增加的负载，才能保持可接受的性能。

图 24-7 中应用程序层只包含一台为客户端请求提供服务的应用程序服务器(AppServer20)。如果该服务器超载，则解决方案的性能将降至不可接受的级别，或变得不可用。

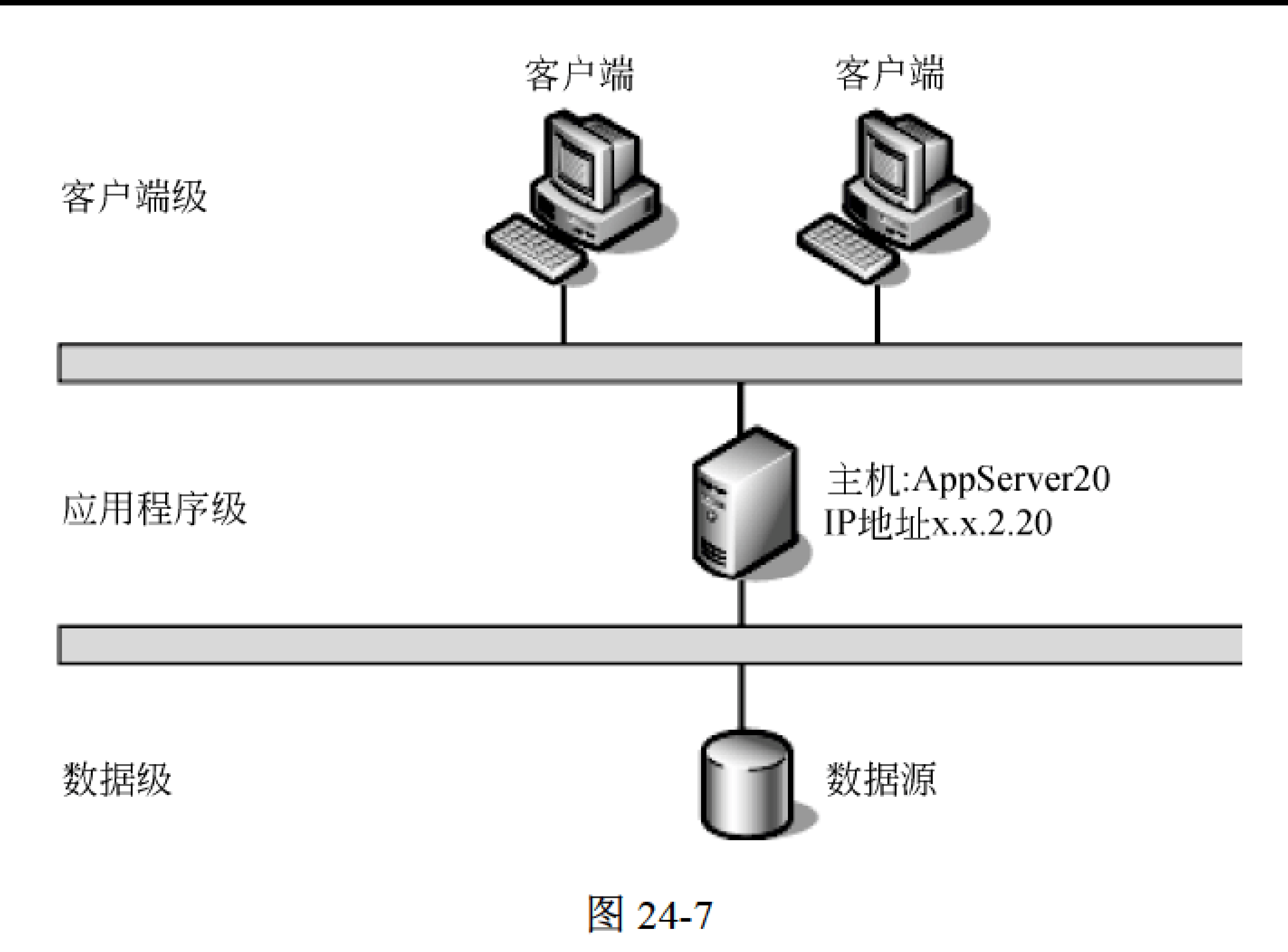


图 24-7

2. 负载平衡层

要提高可伸缩性并保持性能，组织可能会使用负载平衡器来扩展应用程序层。在图24-8中，将两台服务器添加到应用程序层以创建负载平衡集群，该集群将访问数据层数据，并为客户端层中的客户端提供对应用程序的访问服务。

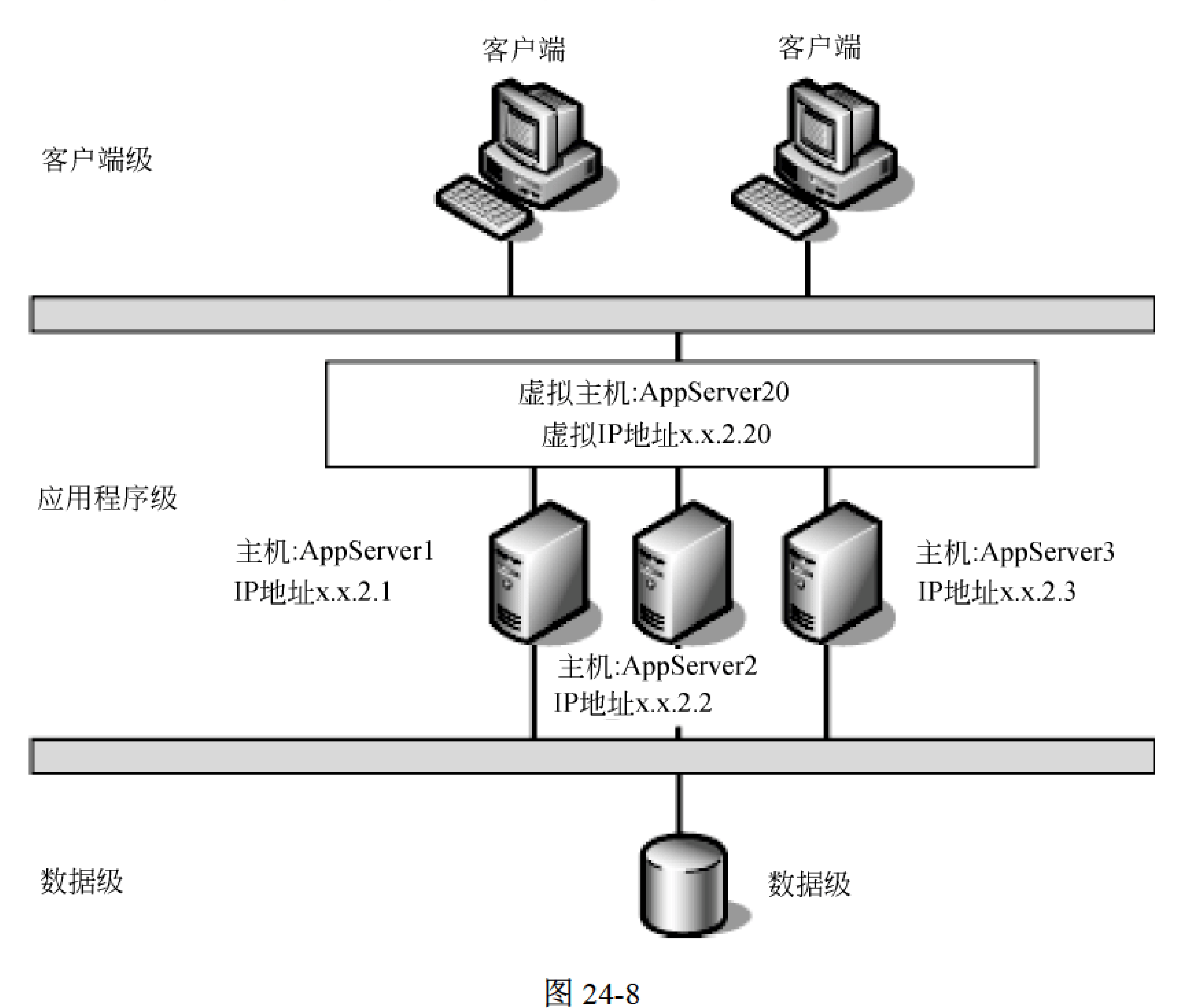


图 24-8

这将得到一个标准的负载平衡设计。硬件设备或运行在主机上的软件将虚拟主机名（AppServer20）和 IP 地址分配给 AppServer1、AppServer2 和 AppServer3。负载平衡的集群向网络公开此虚拟 IP 地址和主机名，并在组内的正常运行服务器之间均衡地分配传入请求

的负载。如果 AppServer1 出现故障，则只需将请求定向到 AppServer2 或 AppServer3 即可，取决于提供此功能的技术，可以将一定数目的额外服务器添加到负载平衡的集群中，以最大限度地提高可伸缩性，并超前满足不断增长的需求。

24.3.10　负载均衡的优缺点

1. 优点

（1）改进的可伸缩性。

可以动态增加部署在 WebLogic Server 集群中的应用程序的容量以满足需要。可以将服务器实例添加到集群中而不会中断服务，应用程序将继续运行而不会影响客户端和最终用户。

（2）更高的可用性。

在 WebLogic Server 集群中，当服务器实例失败时应用程序可以继续进行处理。可通过将应用程序组件部署到集群中的多个服务器实例集群这些组件。这样，如果在其上运行某个组件的服务器实例失败，则将此组件部署到的其他服务器实例可以继续进行应用程序处理。

（3）可能会降低成本。

与更高成本的多处理器系统相比，多台低成本服务器通常会降低成本。

2. 缺点

（1）开发/维护复杂。

如果解决方案必须维护各个事务或用户的状态，则开发负载平衡的解决方案会很困难。

（2）没有解决网络故障问题。

如果在客户端会话过程中服务器或网络出现故障，则可能需要重新登录才能重新验证客户端和重新建立会话状态。

24.4　集群故障转移与复制

24.4.1　Servlet 和 JSP 的复制和故障转移

WebLogic Server 通过复制访问集群 Servlet 和 JSP 的客户端的 HTTP 会话状态，为 Servlet 和 JSP 提供集群支持。对于 Servlet 和 JSP 在集群中的自动复制和故障转移，WebLogic Server 支持两种用于保留 HTTP 会话状态的机制。

（1）硬件负载平衡器。对于使用受支持硬件负载平衡解决方案的集群，负载平衡硬件只是将客户端请求重定向到 WebLogic Server 集群中的任何可用服务器，而集群自身则从集群中的次级服务器获得客户端 HTTP 会话状态副本。

（2）代理插件。在使用 Web 服务器和 WebLogic 代理插件的集群中，代理插件会以透

明方式处理到客户端的故障转移。如果某个服务器失败，该插件则会在刺激服务器上定位复制的 HTTP 会话状态，然后相应地重定向客户端请求。

1. HTTP 会话状态复制

（1）内存中复制。主服务器在客户端首先连接的服务器上创建主会话状态，在集群中的另一个 WebLogic Server 实例上创建次级副本。该副本总是保持最新状态，以便承载 Servlet 的服务器失败时可以使用该副本。

（2）基于 JDBC 的持久性。在基于 JDBC 的持久性中，WebLogic Server 使用基于文件的持久性或基于 JDBC 的持久性维护 Servlet 或 JSP 的 HTTP 会话状态。

2. 使用代理访问集群的 Servlet 和 JSP

使用代理访问 Servlet 和 JSP 如图 24-9 所示。

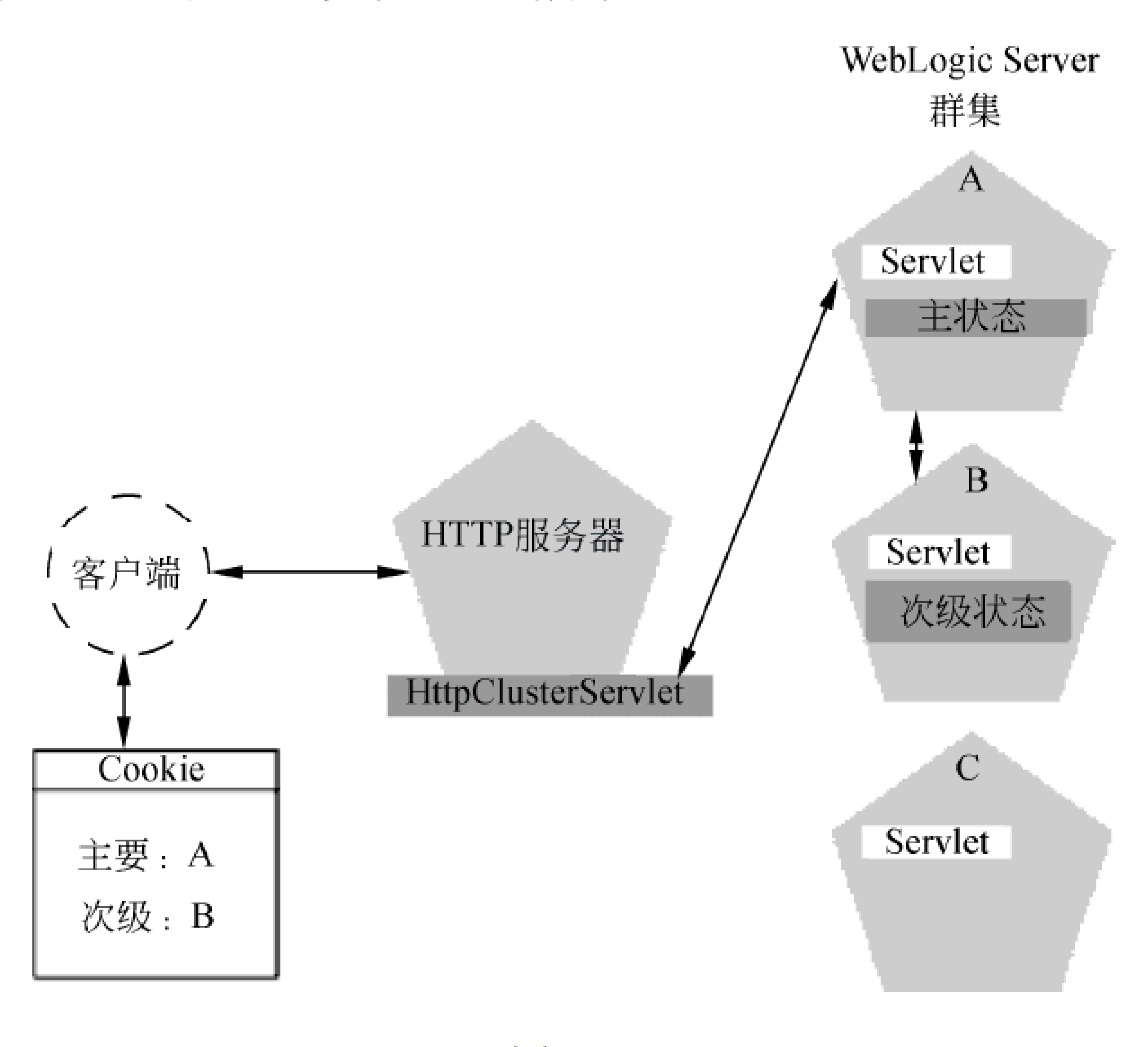


图 24-9

代理故障转移过程如下。

如果主服务器失败，HttpClusterServlet 就使用客户端的 Cookie 信息来确定承载会话状态副本的次级 WebLogic Server 的位置。HttpClusterServlet 会自动将客户端的下一个 HTTP 请求重定向到次级服务器，故障转移对于客户端是透明的。

发生失败之后，WebLogic Server B 成为承载 Servlet 会话状态的主服务器，并且会创建新的次级服务器。在 HTTP 响应中，代理会更新客户端的 Cookie 来反映新的主服务器和次级服务器，以考虑后续故障转移的可能性。

在由两个服务器组成的集群中，客户端将以透明方式故障转移到承载次级会话状态的服务器上。但是，客户端会话状态的复制不会继续，除非另一个 WebLogic Server 变为可用状态并加入该集群。例如，如果原始主服务器重新启动或重新连接网络，则会使用它来承载次级会话状态。

3．使用负载平衡硬件访问集群的 Servlet 和 JSP

客户端通过负载平衡器访问集群的连接过程如图 24-10 所示。

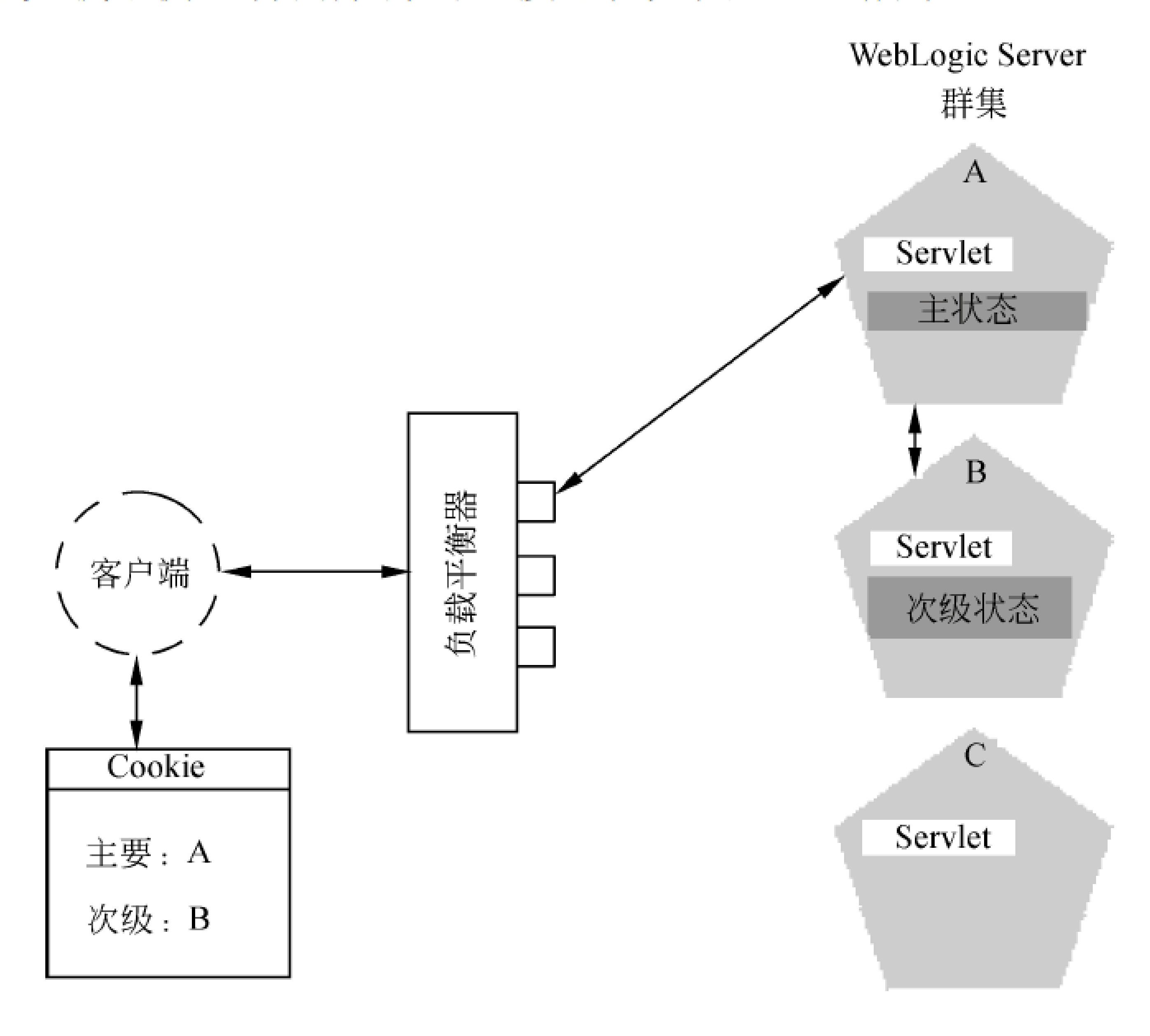


图 24-10

使用负载平衡硬件的故障转移如下。

如果在客户端会话期间服务器 A 失败，则客户端到服务器 A 的下一个连接请求也将失败，如图 24-11 所示。

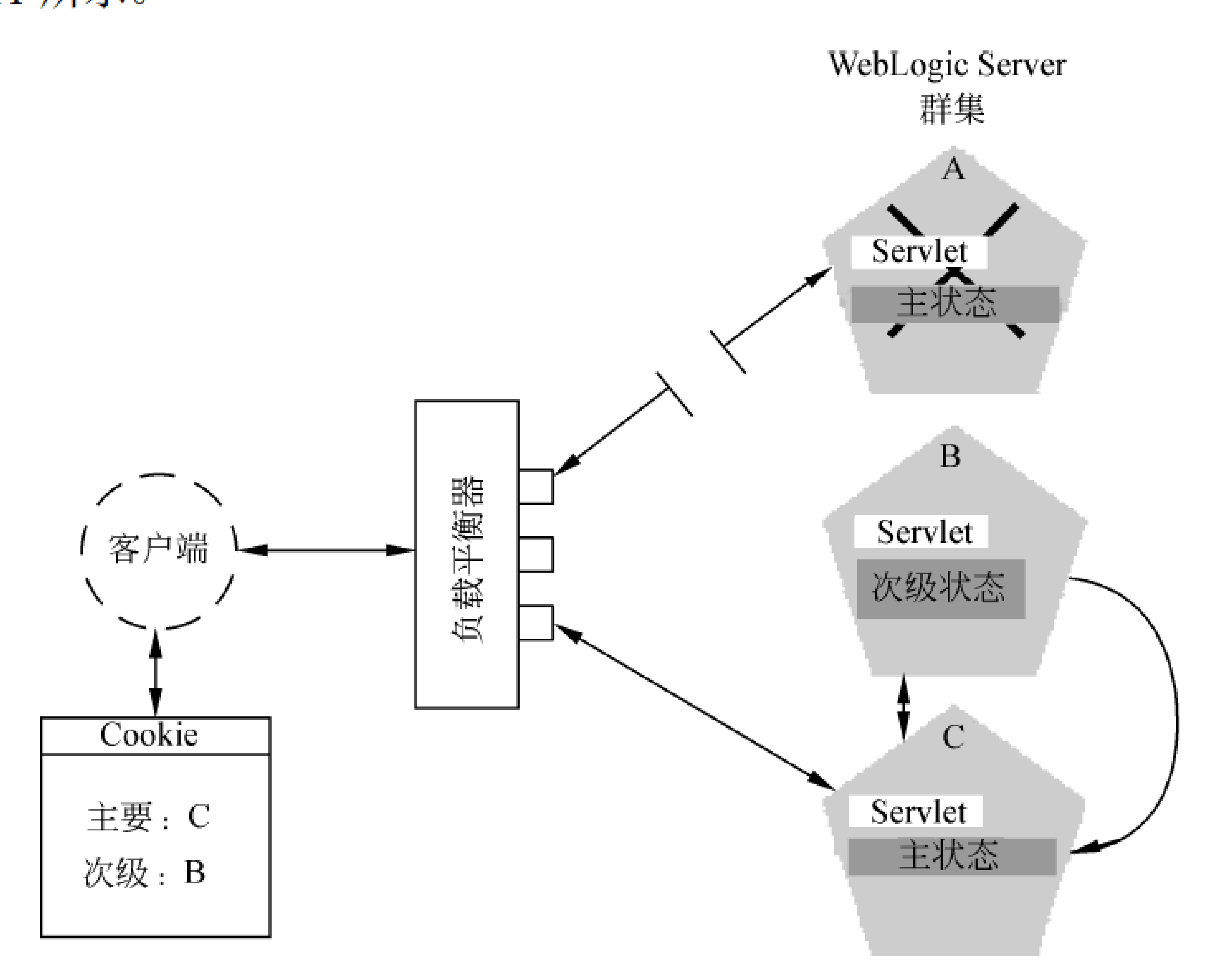


图 24-11

为了响应连接失败，则采取以下措施。

- 负载平衡硬件使用其配置的策略将请求定向到集群中的可用 WebLogic Server。在上面的示例中，假设在 WebLogic Server A 失败之后，负载平衡器将客户端的请求路由到了 WebLogic Server C。
- 当客户端连接 WebLogic Server C 时，服务器使用客户端 Cookie 中的信息（如果使用 URL 重写则使用 HTTP 请求中的信息）来获取 WebLogic Server B 上的会话状态副本。该故障转移过程对于客户端一直保持完全透明。

WebLogic Server C 成为客户端主会话状态的新主机，WebLogic Server B 继续承载会话状态副本。

24.4.2 EJB 和 RMI 的复制和故障转移

可使用可识别副本的存根控件（可在集群中定位对象实例）处理 EJB 和 RMI 对象故障转移。当客户端通过可识别副本的存根控件向失败的服务做出调用时，该存根控件可检测故障并在另一副本上重试此调用。

对于集群对象，自动故障转移通常仅在该对象具有幂等性的情况下才会发生。如果任何方法都可以多次调用，并且每次调用该方法都会产生不同的结果，则该对象具有幂等性。没有永久性副作用的方法就总是符合这样的要求。一定要使用幂等性编写具有副作用的方法。

1. 使用副本感知存根控件集群对象

如果 EJB 或 RMI 对象进行了集群，则该对象的实例将部署在集群中的所有 WebLogic Server 实例上。客户端可以选择要调用该对象的哪个实例。该对象的每个实例都称为一个副本。

支持在 WebLogic Server 中集群对象的关键技术为副本感知存根控件。当您编译支持集群的 EJB 时，appc 会通过 rmic 编译器传递 EJB 的接口以生成该 bean 的副本感知存根控件。对于 RMI 对象 ，您要通过对 rmic 使用命令行选项的方式来显式生成的副本感知存根控件。

副本感知存根控件看起来好像是与常规 RMI 存根控件相同的调用者。但是该存根控件不表示单个的对象，而表示一个副本集合。副本感知存根控件包含了在部署对象的任何 WebLogic Server 实例上定位 EJB 或 RMI 类所需的逻辑。当您部署可以识别集群的 EJB 或 RMI 对象时，其实现将绑定到 JNDI 树中。集群的 WebLogic Server 实例具有更新 JNDI 树的功能，以便列出可在其中使用该对象的所有服务器实例。当客户端访问集群对象时，实现会被副本感知存根控件所替换，该存根控件会被发送到客户端。

该存根控件包含了用于对对象的方法调用进行负载平衡的负载平衡算法（或调用路由类）。对于每个调用，存根控件都可以采用负载算法以选择要调用哪个副本。这就以对于调用者透明的方式在集群中提供了负载平衡，要了解可用于 RMI 对象和 EJB 的负载平衡算法。如果在调用期间发生失败，存根控件会截获到异常，并在另一个副本上重试该调用，这样就提供了对于调用者也透明的故障转移。

2. 有状态会话 EJB 的故障转移

EJB 与普通的 RMI 对象不同，其不同之处在于每个 EJB 都可能生成两个不同的副本感知存根控件：一个用于 EJBHome 接口，一个用于 EJBObject 接口。这就表示 EJB 可以在两个级别实现负载平衡和故障转移。

（1）当客户端使用 EJBHome 存根控件查找 EJB 对象时。

（2）当客户端使用 EJBObject 存根控件针对 EJB 进行方法调用时。

如果主服务器失败，客户端的 EJB 存根控件会自动将进一步的请求重定向到次级 WebLogic Server 实例。此时，次级服务器会使用复制的状态数据创建新的 EJB 实例，并且在次级服务器上继续处理实例。

故障转移之后，WebLogic Server 会选择新的次级服务器来复制 EJB 会话状态（如果集群中还有另一个服务器可用的话）。新主服务器实例和次级服务器实例的位置将在下一次方法调用时在该客户端副本感知存根控件中自动更新，如图 24-12 所示。

图 24-12

注：由于 WebLogic 集群相关部分较多，下面依次在后续三个章节中逐一进行细致介绍。

第 25 章　组播错误分析

组播在集群中的服务器来广播服务可用性信息和心跳信号检测消息时，会因为一些常规配置错误、WLS 版本差异、物理网络问题等一系列原因而导致组播错误，从而影响用户的正常使用。

25.1　组播错误概述

25.1.1　组播的错误表现形式

组播错误的模式多种多样，但通常表现为以下几方面。

- 服务器监听组播地址时出现问题，如发生套接字接收错误。
- 服务器接收组播消息时出现问题，如发生套接字发送错误。
- 或服务器在其他情况下使用组播时失败，如无法创建集群的组播套接字。

25.1.2　组播错误消息

组播常见的错误消息如下。

示例 25-1：

```
<Error> <Cluster> <Multicast socket receive error:
java.io.InterruptedIOException: Receive timed out>
java.io.UnknownHostException
java.io.IOException: Too many open files
Unable to send service announcement
```

25.2　组播错误的成因

组播错误的成因可能是由下列各项之一造成的。

- 常规配置错误。
- WLS 版本组播差异。
- 物理网络问题。
- 打开的文件描述符过多。
- 组播超时。

- ❑ 集群心跳信号检测和组播风暴问题。
- ❑ 多宿主问题。

25.3　组播问题探究

下面将主要阐述几种比较常见的问题，并给出其检查核对方式。

1. 配置错误

- ❑ 检查是否有错别字、拼写错误或 IP 地址或端口号中的数字顺序是否颠倒。
- ❑ 检查组播 IP 地址是否在以下有效范围内：214.0.0.0 和 229.245.245.245 之间。
- ❑ 检查 WLS Administration Console 中的值是否正确。
- ❑ 检查域的配置文件 config.xml 中是否有错误。

2. WLS 版本组播差异

- ❑ 在 WLS 6.1 中，服务器的监听端口用做组播端口。
- ❑ WLS 7.1 和更高版本中配置了单独的组播端口，不再使用服务器的监听端口作为组播端口。
- ❑ 集群中的服务器必须都运行相同版本的 WebLogic Server。

3. 物理网络问题

- ❑ 确认没有物理网络连接问题。
- ❑ 确认没有其他应用程序或集群正在使用同一组播地址。
- ❑ 确认没有将组播 IP 地址分配给多台计算机。
- ❑ 确认组播端口当前未在使用，也未处于等待状态。
- ❑ 使用各操作系统提供的专用命令进行探查，如使用 netstat 或 ping。

4. 打开的文件描述符过多

- ❑ 套接字连接需要文件描述符，该描述符由操作系统（Operating System, OS）分配。
- ❑ OS 资源限制控制可以使用的文件描述符总数；单个进程最多可以打开的描述符数。
- ❑ 确定当前使用的文件描述符的数量。Windows 系统：handle 或 Process Explorer; UNIX 平台：lsof。
- ❑ 有关更多详细信息，可参阅打开的文件过多支持模式。

5. nsswitch.conf 配置

- ❑ “系统数据库”和“名称服务交换机”配置文件：定义使用服务获取信息（如主机名、密码和组）的顺序；位于/etc/nsswitch.conf 中。
- ❑ 如果发生 UnknownHostException，可检查下列参数设置：hosts: files DNS NIS。

6. 组播超时

- 网络接口卡（Network Interface Card, NIC）发生 Failover 时，就可能导致组播超时。
- 尝试禁用 NIC Failover。
- 将组播生存时间设置为更高的值，例如 MulticastTTL=32。

7. 组播超时和 IPMG

- Internet 组管理协议（Internet Group Management Protocol, IPMG）用于在组播组中建立成员资格。
- 使用 IPMG snooping 时，某组的组播通信量只会转发给该组成员的端口。
- 可尝试禁用被管交换机的 IGMP snooping 设置，即 igmp snooping=disable。
- 在 Windows 2000 上，可设置 IPMGLevel=2，以确保 IPMG 组播处理正确。

8. 集群心跳信号检测问题

- 各集群服务器使用组播相互广播常规心跳信号检测消息。
- 通过监视这些心跳检测信号，集群服务器可以确定集群中的某个服务器发生故障的时间。
- 为避免操作系统缓冲区溢出，可设置适当的 Multicast Send Delay 值。
- 还要将操作系统 udp 缓冲区大小参数与 WLS 设置进行比较，检查接收和发送组播信息包大小，即 udp_xmit_hiwat 和 udp_recv_hiwat，确保 Multicast Buffer Size 的值较小。

9. 集群组播风暴

- 组播风暴是指在网络中重复发送组播信息包。
- 如果集群服务器处理消息的速度不够快，就会导致网络通信量增加，如 NAK 和重复发送心跳检测信号。
- 增大组播缓冲区大小有助于加快通告的发送和接收速度，从而防止组播风暴的发生。

10. 设置 Multicast Buffer Size

- 组播缓冲区大小由 TCP/IP 内核参数 udp_max_buf 定义；可以通过 UNIX ndd 实用程序进行设置。
- 不过，在更改组播缓冲区大小时需谨慎行事。

① 请先阅读 Sun 发布的有关此参数的警告，详细信息请参阅 http://docs.sun.com/?p=/doc/ 8066779/ 6jfmsfr70&。

② 每次增加 32KB，并观察更改的效果。

③ 如果仍会发生组播风暴，请再次增加。

④ 继续监视更改的效果和影响。

11. 多宿主配置

- 多宿主是指一台计算机有多个 IP 地址，例如：① 将多个 IP 地址分配给一台物理

计算机；② 一台计算机上有多个网络接口卡。

- ❑ 在 WLS Administration Console 中配置下列参数：① 每个 IP 地址的逻辑计算机名称配置操作步骤为：执行“计算机”→“配置新的 UNIX Machine”→“名称”命令。② 每台服务器的正确接口地址配置操作步骤为：执行 myServer→“配置”→“集群”→“接口地址”命令。

25.4　组播的测试和调试

25.4.1　组播测试

集群中的各服务器共用同一个专用组播地址进行相互间的通信。

在 WLS 中，将在不发送通知的情况下删除杂散（即未与集群关联）的组播消息。

1．组播测试实用程序

utils.MulticastTest 实用程序如下。

- ❑ 显示下列信息给 stdout：① 服务器发送或接收的每条消息的详细信息；② 顺序错误警告或缺失消息警告。
- ❑ 应在需要进行组播测试的每个节点上启动。
- ❑ 要求集群中的每个服务器都具有一个唯一的名称。
- ❑ 可用于检查是否发生了交叉集群组播。

2．组播测试故障排除

- ❑ 如果组播测试失败：① 确认使用了 WLS 所要求使用的 Primary 地址；② 检查是否正确配置和使用了 DNS。
- ❑ 在多宿主环境中：① 使用 ipconfig 获得计算机的 MAC 地址；② 每台计算机的 MAC 地址都应该是唯一的。
- ❑ 在 Solaris 系统上，如果 MAC 地址不是唯一的：① 使用一个接口卡将所有多宿主 Solaris 计算机连接在一起。② 或添加另一个接口卡。

25.4.2　组播调试

所有其他方法均失败时，可使用此方法。

- ❑ 可将许多消息写入日志文件。
- ❑ 涉及以下 3 个参数：DebugCluster、DebugClusterHeartbeats、DebugClusterFragments。
- ❑ 可在服务器启动时静态启动，例如 Dweblogic.debug.DebugCluster=true。
- ❑ 可以使用以下命令动态启动 weblogic.Admin。

25.5 组播排除策略

- ❑ 确保组播通信可以于正在使用的网络地址上传播：① 使用 WebLogic 随附的组播测试工具测试计算机间的组播通信量；② 与网络团队合作，确保组播通信向要求的传送方向。
- ❑ 对网络进行监视，留意异常的通信模式。
- ❑ 根据需要调整缓冲区大小等 WebLogic 设置。
- ❑ 启用组播调试，获取更多信息。

第 26 章　使用代理插件时的 HTTP 负载平衡不均故障

26.1　回顾：常见的代理插件

26.1.1　Apache 代理插件

1. 概述

建站时，考虑到为了分担系统负担以及提高系统安全等原因，需要将 WebLogic 的 HTTP Server 分离，而使用 Apache HTTP Server，这需要安装 Apache-WebLogic 插件来实现。当客户端请求静态页面时，使用 Apache Server，而当客户端请求动态页面时，Apache Server 通过插件来使用 WebLogic Server；对于客户端来说，WebLogic Server 是不可见的，但是却能通过插件来使用 WebLogic Server 的所有服务。

2. 安装插件

在 weblogic/lib/linux 目录（不同的操作系统对应不同的目录）下，有一个共享目标文件 mod_wl.so，这个文件需要安装到 Apache 中，所以要求 Apache 必须支持 DSO（Dynamic Shared Object）；可以通过以下命令来检测您的 Apache 是否支持 DSO：

```
Apache_home/bin/httpd -l
```

如果在列出的模块名中有 mod_so.c，那么您的 Apache 已经支持 DSO 模块，否则需要重新编译 Apache，在编译 Apache 时，需要加以下两个参数：

```
./configure --enable-module=so --enable-rule=SHARED_CORE
```

接下来是安装 mod_wl.so 模块，先进入 weblogic/lib/linux（假设您的操作系统是 Linux），用以下命令安装 mod_wl.so：

```
perl APACHE_HOME/bin/apxs -i -a -n weblogic mod_wl.so
```

以上命令执行完后会复制 mod_wl.so 文件到 APACHE_HOME/libexec 目录中，并在 httpd.conf 文件中自动增加一行：

```
LoadModule weblogic_module libexec/mod_wl.so
```

另外还可以在 Apache 的 httpd.conf 文件中设置 Apache-WebLogic plug-in 的参数，这些参数要包含在：

```
<IfModule mod_weblogic.c>
# 参数值
</IfModule>
```

中，参数和值之间不能有等号。目前常用的参数如下。

（1）WebLogicHost 域名：设置 WebLogic 主机的名字。

（2）WebLogicPort 端口号：设置 WebLogic 主机的端口号。

（3）ConnectTimeoutSecs 秒数：设置服务器连接超时秒数。

（4）ErrorPage URL：如果 Apache 服务器不能请求 WebLogic 服务器，将转到您所设置的错误页面中。

如果不设置以上的参数，也可以通过虚拟主机来实现，具体例子如下：

```
#装载模块
LoadModule weblogic_module libexec/mod_wl.so
```

定位目录，这个目录必须和 WebLogic Server 中放置.jsp 文件的目录一致：

```
<Location />
```

明确指定 WebLogic 模块的操作者：

```
SetHandler weblogic-handler
```

如果请求页面不存在所返回的错误页面：

```
ErrorPage http://www.weblogic-server.com/error.html</Location>
```

客户段请求的动态文件，可增加：

```
MatchExpression *.jsp
```

设置虚拟主机：

```
NameVirtualHost 10.1.1.6
<VirtualHost goldengate.domain1.com>
#WebLogic Server 的主机名和端口
WebLogicHost www.weblogic-server.com
WebLogicPort 7001
#连接超时秒数
ConnectTimeoutSecs 30
</VirtualHost>
```

现在先启动 WebLogic Server 再启动 Apache Server，就可以测试您的 JSP 了。

示例配置如下。

（1）Cluster 环境。

示例 26-1：

```
<IfModule mod weblogic.c>
WebLogicCluster 10.0.1.116:7001,10.0.1.120:7001
MatchExpression /*/cs/*
MatchExpression *.jsp
</IfModule>
<Location /*>
SetHandler weblogic-handler
</Location>
```

（2）单机（Stand Alone）环境。

示例 26-2：

```
<IfModule mod_weblogic.c>
WebLogicHost www.landingbj.com
WebLogicPort 7001
MatchExpression *.jsp
MatchExpression *.do
</IfModule>
```

其他的一些资源，比如图片，js 等就直接用 apache 来负责请求了。

示例 26-3：

```
Alias /uploadfile/ "/home/landingbj/www/uploadfile/"
<Directory "/home/landingbj/www_in/uploadfile/">
#Options Indexes MultiViews
AllowOverride None
Order allow,deny
Allow from all
</Directory>
```

26.1.2　IIS 代理插件

1．概述

IIS 代理服务器插件程序可以代理由 IIS 发往 WLS 的请求，静态页面（HTML）由 IIS 来处理，而动态页面（JSP、Servlet）则由 WLSl 来处理，这样就可以有效地提高服务器的性能。

IIS 和 WLS 可以不必在同一台机器上运行，而对于客户来说，他或她看到的所有请求都好像是由 IIS 来处理的，完全感觉不到 WLS 的存在。IIS 代理服务器插件还实现了可重用请求的连接池和 HTTP1.1 的持续连接机制。持续连接使得服务器每次只打开一个连接来响应客户的多个请求，而不是像通常那样对于页面的不同内容（文字、多个图片）连接多

次。当访问者在一定的时间内没有请求时，插件程序会将此连接保存到连接池中，以便响应客户后来的请求。IIS 代理插件程序通过这种机制以更大程度地提高服务器的性能。至于哪些内容需要通过代理插件程序来处理，您既可以使用基于路径的处理方式，也可以使用基于文件扩展名的处理方式，当然您也可以结合使用这两种方式。基于路径的处理方式只将以指定路径开头的请求交由代理服务器插件处理，比如您将路径指定为/weblogic，这样 http://iss/weblogic/index.jsp 就会转到 http://wls/weblogic/index.jsp，其中 iis 为 IIS 主机，wls 为 WLS 主机。而基于文件扩展名的处理方式，您可以指定所有对特定文件类型的请求都交由代理服务器插件处理，比如您可以指定所有对*.jsp 文件的请求都交给代理服务器插件去处理。

2. 安装插件

（1）将 WLS 目录\bin\iisproxy.dll 复制到 C:\WINNT\system32 目录下。说明：这个 iisproxy.dll 文件放到任何目录内都可以，最好将后面介绍的 iisproxy.ini 这个文件也放到 iisproxy.dll 所在的目录中。

（2）执行“控制面板”→“管理工具”命令，打开“Internet 服务管理器”对话框，在该对话框中右击“默认 Web 站点”图标，在弹出的快捷菜单中执行“属性”命令。接着执行“主目录”→“配置”……→“应用程序映射”→“添加”命令。在“可执行文件”文本框中填入“C:\WINNT\system32\iisproxy.dll”，也可以通过单击浏览按钮来选择此 DLL 文件。在“扩展名”文本框中填入“.jsp”，其他各项保持默认值，然后单击“确定”按钮。这样以后每次遇到对*.jsp 文件的请求时，IIS 都会交给代理服务器处理。您也可以按以上步骤添加其他文件类型。

（3）在 C:\WINNT\system32 目录下建立一个 iisproxy.ini 文件，其内容如下：

```
WebLogicHost=192.168.0.1
WebLogicPort=7001
```

这里的 WebLogicHost 是 WLS 服务器地址，WebLogicPort 是 WLS 服务器端口号。注意：您最好将 iisproxy.ini 文件放到 iisproxy.dll 文件所在的目录下，因为插件程序会按照以下路径来搜索 iisproxy.ini 文件：iisproxy.dll 所在目录、最新版本的 WLS 目录、以前版本的 WLS 目录、C:\weblogic 目录（如果存在）。

（4）以上设置了基于文件扩展名的处理方式，下面设置基于路径的处理方式。将 WLS 目录\bin\iisforward.dll 复制到 C:\WINNT\system32 目录下。执行“控制面板”→“管理工具”命令打开“Internet 服务管理器”对话框，在该对话框中右击“默认 Web 站点”图标，在弹出的快捷菜单中执行“属性”命令。接着执行“ISAPI 筛选器”→“添加”命令，在“可执行文件”文本框中填入“C:\WINNT\system32\iisforward.dll”，也可以单击“浏览”按钮来选择。在“筛选器名字”文本框中填入一个好记的名字，比如 iisforward，然后单击“确定”按钮。按照步骤（2）中的方法，注册一个由 iisproxy.dll 处理的新类型.wlforward。在 iisproxy.ini 文件中再加入下面两行：

```
WlForwardPath=/weblogic
PathTrim=/weblogic
```

WlForwardPath 指定了需要处理的路径字符串，比如对于 http://iis/weblogic/machine/请求会映射到 http://wls/weblogic/machine/，而 http://iis/machine/会被 IIS 直接处理，并不会交给代理服务器插件程序。PathTrim 是去掉请求路径中的/weblogic 字符串，如上面的请求最后会变成 http://wls/machine/。

注 意

每次修改 iisproxy.ini 文件后，都应该执行“控制面板”→“管理工具”命令打开“Internet 服务管理器”对话框，重新启动 IIS Admin Service。下面测试一下，在 WLS 的默认发布目录（一般是 mydomain\applications\DefaultWebApp）下放一个 test.jsp 文件，然后启动 WLS 和 IIS，在浏览器中通过 IIS 地址和端口直接访问这个 test. jsp 文件（如 http://192.168.0.1/test.jsp，IIS 地址 192.168.0.1，端口 80），然后再加上请求路径访问 test.jsp 文件（如 http://192.168.0.1/weblogic/test.jsp），如果能正确显示 test.jsp 的内容，那么 IIS 代理服务器插件安装成功。

26.2　使用代理插件的 HTTP 负载平衡不均的症状和成因

26.2.1　负载不均症状

（1）HTTP 请求的负载平衡不均。

（2）频繁或意外的会话 Failover。

（3）HTTP 回应中的意外状态原始码（比如 400、404、500、503 等），无法从后端 WebLogic Server 确定其产生根源或原因。

26.2.2　负载不均成因

（1）配置问题。

（2）网络问题。

26.3　负载不均探查

26.3.1　探查基本步骤

（1）启动调试。

（2）获取 Debug 信息分析。

（3）确认问题所在。

26.3.2　启动调试

1．启用代理调试

通过在代理设定文档中设定 Debug="ALL"启用代理除错。

Apache 插件（httpd.conf）的设定范例如下。

示例 26-4：

```
<Location /mywebapp>
SetHandlerweblogic-handler
WebLogicCluster sol1:8001,sol2:8001,sol3:8003
Debug ALL
DebugConfigInfo ON
WLLogFile /tmp/wlproxy.log
</Location>
```

IIS 插件（iisproxy.ini）的设定范例如下。

示例 26-5：

```
WebLogicCluster sol1:8001,sol2:8001,sol3:8003
Debug ALL
DebugConfigInfo ON
WLLogFile C:\temp\wlproxy.log
```

iPlanet/SunOne 插件（obj.conf）的设定范例如下。

示例 26-6：

```
<Object name="weblogic" ppath="*/mywebapp/*">
Service fn=wl-proxy
WebLogicCluster="sol1:8001,sol2:8001,sol3:8003"
Debug=ALL
DebugConfigInfo=ON
WLLogFile="/tmp/wlproxy.log"
</Object>
```

2. 启用代理桥接器

在代理设定文档中设定 DebugConfigInfo="ON"。

可以利用类似如下形式的 URL 打开代理桥接器调试页面：

```
http://webserver_host:port/path/xyz.jsp?__WebLogicBridgeConfig
```

其中，path 应当是转到插件的路径。

例如，http://myhost:8080/mywebapp/session.jsp?__WebLogicBridgeConfig。

下面是一个代理桥接器页面范例。

示例 26-7：

```
Query String: '__WebLogicBridgeConfig'
This entry is cluster aware.
ClusterID (from obj.conf): "sol1:8001,sol2:8001,sol3:8001"
This is the same list of servers that is configured in proxy configuration
```

```
file
WebLogic Cluster List:
 • Host: '172.18.137.54' Port: 8001 *Primary*
 • Host: '172.18.137.50' Port: 8001 *Secondary*
General Server List:
Also called dynamic server list and contains the server nodes that are known
to be healthy at this point
 • Host: '172.18.137.50' Port: 8001 Status: OK
 • Host: '172.18.137.54' Port: 8001 Status: OK
ConnectRetrySecs : '2'
ConnectTimeoutSecs : '10'
CookieName : JSESSIONID
Debug : 'ALL'
DebugConfigInfo : 'ON'
DefaultFileName :
DynamicServerList : ON
ErrorPage : ''
FileCaching : ON
Idempotent : ON
KeepAliveEnabled : true
KeepAliveSecs : '20'
MaxPostSize : '-1'
MaxSkipTime : '10'
PathPrepend : ''
PathTrim : ''
SecureProxy : 'OFF'
StatPath : false
WLIOTimeoutSecs (old name is HungServerRecoverSecs): '300'
WLSocketTimeoutSecs : '2'
WLLogFile : '/tmp/wlproxy.log'
WLProxySSL : 'OFF'
Runtime statistics:
 • requests: 4
 • successful requests: 4
 • Exception objects created: 0
 • Exception Objects deleted: 0
 • URL Objects created: 1
 • URL Objects deleted: 1
 • connections recycled: 3
 • UNKNOWN_ERROR_CODE exceptions: 0
 • CONNECTION_REFUSED exceptions: 0
 • CONNECTION_TIMEOUT exceptions: 0
 • READ_ERROR_FROM_CLIENT exceptions: 0
 • READ_ERROR_FROM_SERVER exceptions: 0
 • READ_ERROR_FROM_FILE exceptions: 0
```

```
• WRITE_ERROR_TO_CLIENT exceptions: 0
• WRITE_ERROR_TO_SERVER exceptions: 0
• WRITE_ERROR_TO_FILE exceptions: 0
• READ_TIMEOUT exceptions: 0
• WRITE_TIMEOUT exceptions: 0
• UNKNOWN_HOST exceptions: 0
• NO_RESOURCES exceptions: 0
• PROTOCOL_ERROR exceptions: 0
• CONFIG_ERROR exceptions: 0
• FAILOVER_REQUIRED exceptions: 0
• POST_TIMEOUT exceptions: 0
• REQUEST_ENTITY_TOO_LARGE exceptions: 0
Build date/time: Oct 4 200318:00:57
Change Number: 291942
```

26.3.3 调试信息分析

下面是插件处理一个请求（含会话 cookie）及其产生的日志的范例，在接收新请求和返回期间经历 4 个不同的阶段。

1．接收新请求，分析请求报头和会话 cookie（如果出现的话）

示例 26-8：

```
Mon May 10 13:14:40 2004 URI=[/mywebapp/session.jsp]
Mon May 10 13:14:40 2004 Parsing cookie JSESSIONID=Af48B06Xe6BvewE8yCNWKz
62dsiu028ql
O9GvkS41bg3n53RbJ2Z!-2032244160!-457294087
Mon May 10 13:14:40 2004 getpreferredServersFromCookie: -2032244160!
-457294087
Mon May 10 13:14:40 2004 GET Primary JVMID1: -2032244160
Mon May 10 13:14:40 2004 GET Secondary JVMID2: -457294087
Mon May 10 13:14:40 2004 [Found Primary]: 172.18.137.50:38624:65535
Mon May 10 13:14:40 2004 list[0].jvmid: -2032244160
Mon May 10 13:14:40 2004 secondary str: -457294087
Mon May 10 13:14:40 2004 list[1].jvmid: -457294087
Mon May 10 13:14:40 2004 secondary str: -457294087
Mon May 10 13:14:40 2004 [Found Secondary]: 172.18.137.54:38624:65535
Mon May 10 13:14:40 2004 Found 2 servers
Mon May 10 13:14:40 2004 Hdrs from Client:[host]=[slsol4:3862]
Mon May 10 13:14:40 2004 Hdrs from Client:[user-agent]=[Mozilla/5.0 (Windows; U;
Windows NT 5.1; en-US; rv:1.6) Gecko/20040113]
Mon May 10 13:14:40 2004 Hdrs from Client:[accept]=[text/xml,
application/xml,
application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,image/jpeg,
```

```
image/gif;q=0.2,*/*;q=0.1]
Mon May 10 13:14:40 2004 Hdrs from Client:[accept-language]=[en-us,
en;q=0.5]
Mon May 10 13:14:40 2004 Hdrs from Client:[accept-encoding]=[gzip,deflate]
Mon May 10 13:14:40 2004 Hdrs from Client:[accept-charset]=[ISO-8859-1,
utf-8;q=0.7,*;q=0.7]
Mon May 10 13:14:40 2004 Hdrs from Client:[keep-alive]=[300]
Mon May 10 13:14:40 2004 Hdrs from Client:[cookie]=[JSESSIONID=
Af48B06Xe6BvewE8yCNWK
z62dsiu028ql09GvkS41bg3n53RbJ2Z!-2032244160!-457294087]
Mon May 10 13:14:40 2004 Hdrs from Client:[cache-control]=[max-age=0]
```

2. 连接 WebLogic Server 节点并转寄该请求

示例 26-9：

```
Mon May 10 13:14:40 2004 attempt #0 out of a max of 5
Mon May 10 13:14:40 2004 trying connect to PRIMARY '172.18.137.50'/38624
/65535 at
line 1138 for '/mywebapp/session.jsp'
Mon May 10 13:14:40 2004 WLS info : 172.18.137.50:38624 recycled? 1
Mon May 10 13:14:40 2004 URL::sendHeaders(): meth='GET'
file='/mywebapp/session.jsp'
protocol='HTTP/1.1'
Mon May 10 13:14:40 2004 Hdrs to WLS:[host]=[slsol4:3862]
Mon May 10 13:14:40 2004 Hdrs to WLS:[user-agent]=[Mozilla/5.0 (Windows;
U; Windows
NT 5.1; en-US; rv:1.6) Gecko/20040113]
Mon May 10 13:14:40 2004 Hdrs to
WLS:[accept]=[text/xml,application/xml,application/xhtml+xml,text/html;
q=0.9,text/
plain;q=0.8,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1]
Mon May 10 13:14:40 2004 Hdrs to WLS:[accept-language]=[en-us,en;q=0.5]
Mon May 10 13:14:40 2004 Hdrs to WLS:[accept-encoding]=[gzip,deflate]
Mon May 10 13:14:40 2004 Hdrs to
WLS:[accept-charset]=[ISO-8859-1,utf-8;q=0.7,*;q=0.7]
Mon May 10 13:14:40 2004 Hdrs to WLS:[keep-alive]=[300]
Mon May 10 13:14:40 2004 Hdrs to
WLS:[cookie]=[JSESSIONID=Af48B06Xe6BvewE8yCNWKz62dsiu028ql09GvkS41bg3n5
3RbJ2Z!-
2032244160!-457294087]
Mon May 10 13:14:40 2004 Hdrs to WLS:[cache-control]=[max-age=0]
Mon May 10 13:14:40 2004 Hdrs to
WLS:[Proxy-Path-Translated]=[/home/landingbj/iplanet41sp11/docs/mywebapp/
session.jsp]
```

```
Mon May 10 13:14:40 2004 Hdrs to
WLS:[Proxy-Path-Translated-Base]=[/home/landingbj/iplanet41sp1
1/docs]Mon May 10 13:14:40 2004 Hdrs to WLS:[WL-Proxy-Client-IP]=[10.
40.4.45]Mon May 10 13:14:40 2004 Hdrs to WLS:[WL-Proxy-SSL]=[false]
Mon May 10 13:14:40 2004 Hdrs to WLS:[Proxy-Client-IP]=[10.40.4.45]
Mon May 10 13:14:40 2004 Hdrs to WLS:[X-Forwarded-For]=[10.40.4.45]
Mon May 10 13:14:40 2004 Hdrs to WLS:[Connection]=[Keep-Alive]
Mon May 10 13:14:40 2004 Hdrs to WLS:[X-WebLogic-Request-
ClusterInfo]=[true]
Mon May 10 13:14:40 2004 Hdrs to WLS:[X-WebLogic-KeepAliveSecs]=[30]
Mon May 10 13:14:40 2004 Hdrs to
WLS:[x-weblogic-cluster-hash]=[sUuYnuyhFpkhw6ty44bkNczWnaA]
Mon May 10 13:14:40 2004 INFO: sysSend 947
```

3. 读取 WebLogic Server 节点的回应

示例 26-10：

```
Mon May 10 13:14:40 2004 Hdrs from WLS:[Date]=[Mon, 10 May 200417:14:46 GMT]
Mon May 10 13:14:40 2004 Hdrs from WLS:[Server]=[WebLogic Server 8.1 SP2
Fri Dec 5
15:01:51 PST 2003 316284 ]
Mon May 10 13:14:40 2004 Hdrs from WLS:[Content-Length]=[3895]
Mon May 10 13:14:40 2004 Hdrs from WLS:[Content-Type]=[text/html]
Mon May 10 13:14:40 2004 parsed all headers OK
```

4. 将该回应写回用户端

示例 26-11：

```
Mon May 10 13:14:40 2004 Hdrs to client:[Date]=[Mon, 10 May 200417:14:46
GMT]
Mon May 10 13:14:40 2004 Hdrs to client:[Server]=[WebLogic Server 8.1 SP2
Fri Dec 5
15:01:51 PST 2003 316284]
Mon May 10 13:14:40 2004 add content-length to srvhdrs, [Content-Length]=
[3895]
Mon May 10 13:14:40 2004 Hdrs to client:[Content-Type]=[text/html]
Mon May 10 13:14:40 2004 canRecycle: conn=1 status=200 isKA=1 clen=3895
isCTE=0
Mon May 10 13:14:40 2004 closeConn: pooling for '172.18.137.50/38624'
Mon May 10 13:14:40 2004 request [/mywebapp/session.jsp] processed
sucessfully..................
```

26.4　问题排除检查清单

根据调试信息，一步一步排除，看问题出在哪个环节，这样能为解决问题节省时间。

（1）打开 Debug。

- 在代理设定档中设定 Debug="ALL"。
- 在代理设定档中设定 DebugConfigInfo="ON"。

（2）追踪请求的整个过程。

- 接收 WebServer 发出的新请求。
- 连结 WebLogic Server 节点并转寄该请求。
- 读取 WebLogic Server 节点的回应。
- 将该回应写回用户端。

（3）分析 Debug 信息，确定问题出现的环节。

（4）追踪问题源，根据现象解决问题。

第 27 章　HTTP 会话复制失败故障

27.1　回顾：HTTP 会话、持久性和复制

27.1.1　HTTP Session

会话是一种服务器端技术，用来在特定客户端发送的一系列相关联的浏览器请求之间跟踪用户，保存用户状态信息。

27.1.2　HTTP 会话持久性

使会话状态得以永久性存储，一旦 Primary 服务器出现故障，通过它能够实现故障容错和选择性地进行 Failover，可通过若干种方法来执行。

1．内存（单个服务器，不复制）

当您使用基于内存的存储方式时，所有会话信息都存储在内存中，并且当您停止和重新启动 WebLogic Server 时，这些信息将会丢失。

2．文件系统持久性

会话信息存储在指定的 PersistentStoreDir 中的一个文件中。

3．JDBC 持久性

会话信息存储在数据库表中。

4．基于 cookie 的持久性

会话信息存储在 cookie 中。

5．内存中复制（在集群内）

会话数据从一个服务器实例复制到内存中的另一个实例中。

27.1.3　HTTP 会话 Failover

依赖于服务器故障期间维护的会话状态，要求使用下列方法之一持久性存储会话状态。

（1）文件系统持久性；

（2）JDBC 持久性；

（3）内存中复制（在集群内）。

如果使用内存中复制持久性，可以自动执行，因为在 Primary 服务器发生故障时有最新的 Secondary 副本可以使用。

27.2　复制失败的成因和故障症状

27.2.1　HTTP 会话复制失败的故障症状

（1）会话数据丢失和/或会话没有按照预期方式运行，并且会话数据有所丢失。

（2）在会话尚未关闭或超时的情况下要求客户端重新登录。

（3）您在服务器的日志文件中看到与 HTTP 会话失败相关的错误和警告。

（4）没有正确地将请求 Failover 到另一个服务器。

（5）如果未将会话状态从 Primary 服务器复制到 Secondary 服务器，就会发生 HTTP 会话复制失败。

27.2.2　HTTP 会话复制失败的可能成因

会话复制失败通常是因为组播、网络问题引起的。有时候，配置问题也会导致失败。此外，需确保输入到会话中的数据必须是可序列化的，否则复制可能会失败，如下列出了基本的失败成因。

（1）组播故障或网络故障。

（2）集群或 Web 应用程序配置问题。

（3）会话数据不可序列化。

（4）应用程序没有正确使用 HTTP 会话状态。

在下列情况下，将在 Primary 服务器上记录消息。

（1）HTTP 会话无法创建 Secondary 副本。

（2）设置了集群调试标志。

Primary 服务器上的输出示例如下。

示例 27-1：

```
<Nov 6, 2003 12:59:12 PM EST> <Debug> <Cluster> <000000>
<Unable to create secondary for -5165892837402719733>
<Nov 6, 2003 12:59:12 PM EST> <Debug> <Cluster> <000000> <Error creating
econdary 5165892837402719733 on -7957889153726652135S: 210.23.23.1:[9001,
9001,-1,-1,9001,-1,-1]:mydomain:server2>
```

上面的消息意味着会话复制已失败。

JSESSIONID 也会显示为如下形式。

示例 27-2：

```
JSESSIONID=1E9Xwn7nLYfOsc1obgRZIwW5s72an7HPPvSD7iaWHMXzpHga5cQj!
-1587343083!NONE
```

Secondary 服务器散列信息将变为 NONE。

27.3 探查 HTTP 会话复制失败

27.3.1 探查会话复制失败的基本步骤

（1）收集诊断数据：在 Primary 服务器和 Secondary 服务器中均启用集群调试、Failover 调试和复制调试，确保调试输出发送到了 stdout 和服务器日志。

（2）确认是否正确使用和配置了会话数据、持久性和复制；确认 Web 应用程序针对会话复制的配置参数正确无误。

（3）还要检查集群配置的一致性。

（4）检查会话数据是否可序列化。

（5）查找其他网络错误或组播错误。

（6）确保应用程序正确使用会话数据。

27.3.2 启用调试

HTTP 会话活动调试如下

（1）可以使用 weblogic.Admin 命令行实用程序来动态地启用或关闭调试选项。

例如，若要在 ServerDebug Mbean 的所有管理实例（即管理服务器或托管服务器）上启用 DebugCluster：

```
java weblogic.Admin -url t3://localhost:6151 -username system -password
weblogic SET -type ServerDebug -property DebugCluster true
```

（2）另外，对于要调试的每个服务器，可以编辑<ServerDebug/>节中的 config.xml 和 Mbean 要素，将值设置为 true 表示启用，或设置为 false 表示禁用。然后必须重新启动管理服务器。托管服务器将重新连接到管理服务器，然后调试标志使动态生效：

```
<ServerDebug DebugCluster="true" Name="myserver"/>
```

（3）在设置了所有标志后，在 config.xml 的末尾处，ServerDebug 标记将类似于如下形式：

```
<ServerDebug ClassFinder="true" DebugCluster="true" DebugClusterAnnounc-
ements="true" DebugFailOver="true" DebugReplication="true" DebugReplica-
tionDetails="true" Name="MyServer1"/>
```

确保服务器的 stdOutSeverity 级别为 INFO，且 StdoutDebugEnabled 被设置为 true。调试信息将被记录到服务器日志以及标准输出中。

27.3.3　调试信息分析

1. 创建新 HTTP 会话示例

创建新 HTTP 会话时，服务器会记录 Primary 会话和 Secondary 会话的位置。

Primary 服务器上的输出示例如下。

示例 27-3：

```
<Oct 9, 2003 12:38:21 PM PDT> <Debug> <Cluster> <000000>
<Creating primary 5165892837402719733>
<Oct 9, 2003 12:38:21 PM PDT> <Debug> <Cluster> <000000> <Created secondary
for 5165892837402719733 on -7957889153726652135S: 210.23.23.1: [9001,9001,
-1, -1,9001, -1, -1]: mydomain: Server2>
```

Secondary 服务器上的输出示例如下。

示例 27-4：

```
ExecuteThread: '1' for queue: 'Replication'> <kernel identity> <> <000000>
<Creating secondary 5165892837402719733>
####<Oct 9, 2003 12:38:21 PM PDT> <Debug> <Cluster> <machine1-c840> <server2>
<ExecuteThread: '1' for queue: 'Replication'> <kernel identity> <> <000000>
<Updated local secondary of 5165892837402719733>
```

2. HTTP 会话更新示例

更新 HTTP 会话时，在 Primary 服务器和 Secondary 服务器上均会记录消息。

Primary 服务器上的输出示例如下。

示例 27-5：

```
<Oct 9, 2003 12:38:21 PM PDT> <Debug> <Cluster> <000000>
<Updated remote secondary for 5165892837402719733>
```

Secondary 服务器上的输出示例如下。

示例 27-6：

```
####<Oct 9, 2003 12:38:21 PM PDT> <Debug> <Cluster> <machine1-c840> <server2>
<ExecuteThread: '1' for queue: 'Replication'> <kernel identity> <> <000000>
<Updated local secondary of 5165892837402719733>
```

JSESSION 的形式如下。

示例 27-7：

```
JSESSIONID=1E9Xwn7nLYfOsclobgRZIwW5s72an7HPPvSD7iaWHMXzpHga5cQj!
-1587343083!-1587348922
```

JSESSIONID 是默认的 cookie 名称，可以在 weblogic.xml 中将其更改为任何内容。

JSESSIONID 的格式如下。

示例 27-8：

```
SessionId!PrimaryServer JVM Hash!SecondaryServer JVMHash
```

27.3.4 问题排查

1. 探查内存（单个服务器，不复制）

（1）当您使用基于内存的存储方式时，所有会话信息都存储在内存中，并且当您停止和重新启动 WebLogic Server 时，这些信息将会丢失。

（2）确保您在运行 WebLogic Server 时已分配足够的堆大小；否则，您的服务器可能在大量负载情况下用尽内存。

（3）不是集群配置的推荐类型（因为数据保存在堆中，且不可用于任何其他服务器）。

（4）检查是否只通过代理服务器或硬件负载平衡器访问服务器，如果使用硬件负载平衡器，则它必须支持兼容的被动或主动 cookie 持久性机制以及 SSL 持久性。

2. 文件系统持久性

（1）确认已在 weblogic.xml 中正确指定了 WebLogic Server 存储会话的目录。您还必须自行创建此目录，并确保已分配访问此目录的适当权限。

（2）确保拥有足够的磁盘空间。

3. JDBC 持久性

确保连接到数据库的连接池拥有对所用数据库表的读/写权限。

4. 探查基于 cookie 的持久性

（1）确保在 HTTP 会话中没有存储 Java、LANG、String 以外的任何内容。

（2）不要刷新您的应用程序代码中的 HTTP 响应对象。

（3）确保响应的信息长度超过所设置的缓冲区大小（默认值为 8192 字节）。

（4）确保在浏览器中启用了 cookie。

（5）确保在使用基于 cookie 的会话持久性时，没有在字符串中使用逗号。

5. 检查应用程序的 weblogic.xml 文件中的会话持久性类型参数

使用内存中复制的 weblogic.xml 示例如下。

示例 27-9：

```
<session-descriptor>
        <session-param>
                <param-name>
```

```
            PersistentStoreType
        </param-name>
        <param-value>
                replicated
        </param-value>
    </session-param>
</session-descriptor>
```

6．探查可序列化会话数据

为了支持 HTTP 会话状态的内存中复制，所有 Servlet 和 JSP 会话数据都必须是可序列化的，否则会话复制将会失败。当启用了调试标志时，WebLogic Server 将在下面输出警告消息，指示会话仍未被复制。您必须使该对象变为可序列化的对象，这样才能复制它。其他对象的会话复制将会正常进行。

调试消息如下。

示例 27-10：

```
<Oct 8, 2003 2:10:45 PM PDT> <Error> <Cluster> <000126> <All session objects should be serializable to replicate. Please check the objects in your session. Failed to replicate non-serializable object>
```

解决办法：找到从中抛出错误的页面，并确保输入会话中的所有数据是可序列化的。

7．检查网络/组播问题

确保网络是完好的，且没有组播问题。您可以执行组播测试来确保组播 IP 工作正常。运行 utils.MulticastTest 实用程序，语法形式类似于：

```
java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]
```

8．验证集群配置

从集群列表中选择 Primary 服务器和 Secondary 服务器。在一个由两个服务器组成的集群中，如果该集群没有包含所有服务器，则不能选择 Secondary 服务器，从而导致会话数据不能被复制。

若要验证，可执行下列命令。

（1）确保 weblogic.jar 在类路径中。

（2）若要获得集群中的所有服务器：

```
java weblogic.Admin -username weblogic -password weblogic -url http://oneo-fthemanagedserverurlinthecluster:6151/ GET -type ClusterRuntime .pretty
```

这样将列出集群中的所有服务器。可以将 URL 改变为集群中的每个服务器，以确保它们拥有相同的条目。

9．应用程序代码诊断

确保仅在应用程序代码中使用 HttpSession 中的 setAttribute/removeAttribute 方法来更新

HTTP 会话。如果您使用其他设置方法来更改会话内的对象，WebLogic Server 将不复制这些更改。

不要使用 HTTP 会话的 putValue 和 removeValue 方法，因为它们不受支持，并且当您在应用程序中使用这些方法时，可能会出现会话数据复制问题。相反，应仅使用 HttpSession 的 setAttribute/removeAttribute 方法。

10. cookie 与 URL Rewriting

在某些情况下，浏览器或无线设备可能不接受 cookie，这样会使利用 cookie 的会话跟踪不能进行。当 WebLogic Server 检测到浏览器不接受 cookie 时，URL Rewriting 是对这种情况的一个可自动替换的解决方法。

通过设置 WebLogic-specific 部署描述符 weblogic.xml 中、<session-param>元素下的 URLRewritingEnabled 属性，在 WebLogic Server 中启用 URL Rewriting。此属性的默认值为 true。

27.4 HTTP 会话性能因素

影响 HTTP 会话性能的其他因素如下。

（1）序列化开销是花费在将会话数据转换为可写入格式上的开销将随会话数据量的增加而增加。

① 会话数据量大时，应考虑采用 JDBC 持久性或基于文件的持久性，以降低网络通信量或内存使用量；不过，由于需要访问外部资源，JDBC 持久性或基于文件的持久性的运行速度可能会降低。但由于这些开销与会话数据大小有关，所以应避免复制大量数据！

② 会话数据量小时，可以充分利用基于 cookie 的会话持久性，因此，不需要采用集群 Failover 逻辑，WLS 应用程序的管理会更简单。

（2）多帧访问会话数据时，表示可能为一个客户端创建多个会话，将不会在帧集内同步请求，会话更新可能会导致会话数据不一致，为确保一致性，需要小心处理会话数据。

27.5 故障排除清单

27.5.1 收集诊断数据

（1）在 Primary 服务器和 Secondary 服务器中均启用集群调试、Failover 调试和复制调试。

（2）确保调试输出发送到了 stdout 和服务器日志中。

27.5.2　确认配置

（1）确认是否正确使用和配置了会话数据、持久性和复制。

（2）确认 Web 应用程序针对会话复制的配置参数正确无误。

（3）还要检查集群配置的一致性。

（4）检查会话数据是否可序列化。

（4）查找其他网络错误或组播错误。

（5）确保应用程序正确使用会话数据。

第 28 章　类转换异常故障

28.1　回顾：Java 类、转换和类加载器

Java 类是描述具有共同属性的一组对象的规范，用做创建类实例（类的特定副本）的模板，通常包括属性或实例变量，描述对象的特征还有方法，通过方法可以对类实例执行操作。类可以扩展另一个类，也就是从一个基类或超类继承属性和方法，以及根据需要添加其他属性和方法。类是 Java 的核心，由于类定义了一个对象的形式和属性，因此整个 Java 语言建立在这个逻辑结构之上。同样，类构成了 Java 面向对象程序设计的基础。任何希望在 Java 程序中实现的想法都必须封装在类中。在一个类中，程序的作用体现在方法中。

类是对象的模板，对象是类的实例。当您定义一个类时，就声明了该类的确切的形式和属性。通过指定类中包含的数据和对这些数据进行操作的代码来定义一个类。最简单的类可以只包含代码或只有数据，大部分实际的类二者都有。这里使用 Class 关键字来声明一个类。通常定义 Class 的形式如下所示。

示例 28-1：

```
Class classname {
    type instance-varable1;
    type instance-varable2;
    //...
    type instance-varableN;
    type metbodname1(parameter-list) {
    //body of method
}
type metbodname2(parameter-list) {
    //body of method
}
//...
type metbodnameN(parameter-list) {
    //body of method
}
}
```

类转换是指在某些情况下，可以将一个 Java 的实例转换为另一个类的实例，两个类需要位于同一族父子层次结构中，使对象能够进行升级转换以及成为超类的实例；使对象能够进行降级转换，即成为子类的实例，但只有在对象是子类的实例时，才能进行此转换。通常在编译时进行验证，不过，由于某些转换只能在运行时进行验证，因而会导致

ClassCastException。

Java 类加载器是 JVM 的一个组件，用于在运行时查找其他必须的类并将它们加载到内存中，按层级方式组织，根（启动）类加载器由 JVM 自己创建，旨在通过以下方法重用已加载的类：在缓存中保存已加载类的列表，并在可能的情况下重用已加载的类；使用 delegation 模型，及类加载器请求其父类加载器加载必需的类，这样就可以在最高级别加载类并重用类，也可以通过应用程序进行定义，以供自定义类库使用或划分名称空间。

处理类加载器时，须记住以下事项：类在被加载后，永远不能由同一个类加载器重新加载；如果动态重新部署，需要使用新的类加载器；一般来说类加载器始终会先询问其父类加载器，然后再加载类；同级类加载器彼此独立，互不知晓；由不同类加载器加载的同一类文件将被视为不同的类。

28.2　类转换异常的故障症状和成因

类转换异常：在运行时表现为 ClassCastException，通常会终止用户请求，可能会从下列位置抛出：在应用程序内；由 WLS 自身；几乎任何子系统，例如 Web 或 EJB 容器、JCA。

类转换异常的典型成因包括类设计不合理和使用不兼容的类；加载两个版本或两个来源的相同类时，在究竟要使用哪个类的问题上发生混淆；或尽管使用了兼容的类，但这些类使用不同的类加载器加载。

28.3　探查类转换异常

要确定出错位置，可使用异常的输出堆栈跟踪。如果异常则说明由应用程序类导致。如果可以编辑和编译代码，可添加额外的诊断代码来收集更多信息。看似从 WLS 子系统抛出的问题，可先检查是否存在已知的 WLS 问题。

28.3.1　应用程序诊断

应用程序代码诊断示例如下。

示例 28-2：

```
someObject obj = bar.method();
System.err.println("The object " + obj + " class loader is " + obj.getClass().
getClassLoader());
System.err.println("Class Foo class loader is " + Foo.class.getClassLoader());
Foo f = (Foo) obj;        /* line where exception occurs */
```

输出示例如下。

示例 28-3：

```
The object Foo@@3e86d0 classloader is sun.misc.Launcher$AppClassLoader@b9d04
```

```
Class Foo classloader is weblogic.utils.classloaders.ChangeAwareClassLo-
ader@5998cb finder: weblogic.utils.classloaders.MultiClassFinder@7c2528
```

如果不同的类加载器将兼容的类或同一类加载了两次，请检查是否两个类被打包到不同的模块中；是否每个包或模块均由不同的类加载器加载；应用程序是否更改了 WLS 线程的上下文类加载器，例如，通过调用 Thread.setContextClassloader(cl)来更改。

可能需要进行以下工作：重新打包应用程序以确保类兼容性；使用 Thread.setContext Classloader（cl）后重设线程的类加载器。

28.3.2 已知的 WebLogic Server 类转换问题

已知的 WLS ClassCastException 问题：小程序（Applet）缓存加载；JCA connectors；Servlet 动态重新加载；使用 prefer-web-inf-classes 功能；RMI 引用调用与传值调用；集群 HTTP 会话中的 EJB 引用。

访问类加载器资源信息时，如果存在下列情况，就可能会抛出 ClassCastException。

使用了小程序缓存标记，例如，cache_option 或 cache_archive；WLS 内部类是代码基，也就是 codebase=/bea_wls_internal/classes。

解决此问题的方法：使用缓存选项时，不要将/classes（ClasspathServlet）作为代码基。

在下列情况下将抛出 ClassCastException。如果客户端将代理对象转换为实现资源适配器的 Connection 接口的类。此时应先创建一个接口（用于扩展 Connection 类），然后，创建一个实现该接口的类将代理对象转换为该新类。如果资源适配器进行了类似转换，WLS 8.1 SP2 以上的版本会处理异常并返回未包装的 Connection 对象，但某些功能会被禁用，而 WLS 8.1 SP2 之前版本则不处理异常。

使用动态 Servlet 或 JSP 重新加载时，Servlet 由新类加载器重新加载，因此先前加载的类将与使用新类加载器加载的类不兼容，这就会导致 ClassCastException。因此，须确保 Servlet 会话对象可序列化，这样就可以使用新加载的类成功地将它们重新创建为对象。

默认情况下，类加载器使用 delegation 模型确保已加载的类得到重用。不过，对于 Web 应用程序，可通过以下方式覆盖该默认值：将元素 prefer-web-inf-classes 设置为 true，这样 Web 应用程序将使用第三方类的本地版本，而该类可能是 WebLogic Server 的一部分，这意味着，将先加载 Web 应用程序的 WEB-INF 目录类，然后再加载由应用程序类加载器或系统类加载器加载的类。

RMI 调用可以是传值调用，即将参数的副本传递到目标方法；引用调用，即将指向参数的指针传递到目标方法，从而能够直接改动对象。WLS 对 RMI 调用做了优化，在应用程序内使用引用调用；但对于应用程序间的调用，由于可能会发生类加载器问题，仍然使用传值调用。

在集群中，可能会出现下列情况：如果 HTTP 会话包含一个可序列化的自定义对象，而该对象包装了一个 EJBObject，会话复制将导致 ClassCastException，请使用 EJB 句柄，不要使用 EJBObject！如果在集群中单独部署 Web 应用程序和 EJB，且 HTTP 会话包含一个可序列化的自定义对象，而该对象包装了一个 EJB 句柄：会话 Failover 后使用 EJB

句柄将导致 ClassCastException，务必将 Web 应用程序和 EJB 打包在同一个 EAR 文件中！

28.4　故障排除检查清单

排查策略如下：

（1）使用 ClassCastException 堆栈跟踪对问题进行初步诊断。

（2）如果应用程序代码为问题所在：

① 添加诊断代码以收集类和类加载器的详细信息；

② 解决任何影响类加载的打包问题。

（3）如果问题出在 WLS 子系统中：

① 查看 ClassCastException 已知成因列表，使用建议的解决办法解决问题；

② 对于无法确定的问题，可与 BEA 技术支持部门联系。

（4）继续监视，看是否还会发生问题。

第 29 章　SSL 问题故障

29.1　SSL 相关知识

29.1.1　什么是 SSL

SSL（Secure Sockets Layer，全称安全套接字层）具体含义如下。

- 是一种用于确保应用程序间数据传输的机密性和可靠性的 Internet 传输层技术。
- 使各应用程序可以通过公钥加密相互验证身份。
- 使用密钥加密和数字签名为应用程序间交换的数据加密。
- 它可能是单向的，即服务器向客户端验证身份；也可能是双向的，即同时客户端也向服务器验证身份。
- 在初次建立连接时需要握手。

29.1.2　什么是 SSL 证书、证书链

SSL 证书：用于以数字方式建立组织的 Credential 可以自签名，不过自签名证书自身不具有身份验证的价值。

它包括以下几部分。

- 组织的名称和 IP 地址。
- 组织的公钥。
- 证书颁发机构（Certificate Authority, CA）名称以及 CA 的数字签名。

SSL 证书链是指一系列证书。

- 每个证书都包含该证书的签发者的名称，即链中下一个证书的请求者名称。
- 每个证书都使用其颁发者的私钥来签名，该签名可以使用证书链中下一个证书颁发者的公钥进行验证。
- 根证书颁发机构的证书（最后一个证书）是自签名证书。

29.1.3　证书类型

主要的证书类型如下。

- 保密性增强邮件（Privacy Enhanced Mail, PEM）可以：① 包含所有私钥和公钥

（RSA/DSA）及（x509）证书；② 用于文本模式传输。

- ❑ 辨别编码规则（Distinguished Encoding Rules, DER）可包含所有私钥、公钥和证书；是大多数浏览器的默认格式。
- ❑ 公钥加密标准 #12（Public Key Cryptography Standards #12, PKCS#12）可包含所有私钥、公钥和证书，并以二进制格式存储。

29.1.4　证书颁布机构

证书的颁布机构简称 CA。

- ❑ 是验证组织及其站点身份的受信任的第三方。
- ❑ 必须遵循严格的程序来对接受其发放的证书的组织进行身份验证。
- ❑ 向申请公司签发（公钥）数字证书，该证书使用 CA 的私钥签署，以确保可靠性。
- ❑ CA 的公钥广泛分发。

29.1.5　什么是 SSL 握手

SSL 握手含义如下。

- ❑ 是客户端和服务器通过交换消息建立 SSL 会话的过程。它们就会话的安全属性进行协商；在协议版本上达成一致并选择加密算法；使用公钥加密技术生成共享密钥。
- ❑ 涉及服务器时用它的证书向客户端验证自身的身份。
- ❑ 客户端也可能需要向服务器验证自身的身份（采用双向 SSL 时）。
- ❑ 需要客户端和服务器合作以建立信任关系以及创建要在后续会话中使用的密钥。

29.2　SSL 问题概述

29.2.1　WebLogic Server SSL 配置

WebLogic Server 和 SSL 配置如下。

- ❑ WLS 通过专用监听端口（默认端口为 7002）支持 SSL。
- ❑ 要建立 SSL 连接，Web 浏览器需使用 SSL 监听端口和连接 URL 中的 HTTPS schema 连接到服务器：https://myserver:7002。
- ❑ SSL 要求生成公钥和私钥及通过受信任的第三方使用数字签名和数字证书来建立对这些密钥的信任。
- ❑ 证书存储在密钥库，即配置 WebLogic Server 使用的密钥库中。
- ❑ 默认情况下，SSL 为单向，即服务器向客户端验证身份，但也可以将 WLS 配置为双向，即客户端也必须向 WLS 服务器验证身份。

29.2.2 配置 WLS 密钥库

配置 WebLogic Server 以使用密钥库。

- 选择 Server→my Server→Configuration 选项卡→Key stores & SSL 选项卡。
- 选择所需的身份密钥库和信任密钥库的类型，即 Custom Identity 和 Custom Trust。
- 对于每一项，定义密钥库文件名的名称、密钥库类型和密码，例如 my Key Store、JKS 和 my JKS Password。
- 然后为 SSL 身份定义私钥别名（例如，my Server Identity），并定义私钥和服务器数字证书的存储机制。
- 接下来为 Trust 定义受信任 CA 文件的存储机制。
- 记得启用 SSL 监听端口，例如 7002。
- 最后，重新启动服务器以使更改生效。

29.2.3 SSL 问题成因

SSL 问题可能会出现在下列情形中。

- 客户端或服务器 SSL 配置。
- 证书的格式或内容。
- SSL 软件本身。

最常见的 SSL 故障如下。

- 证书过期失效或不正确。
- 证书链问题。
- 主机名验证失败。
- 握手失败。

29.2.4 SSL 问题的故障症状

大多数 SSL 问题出现在连接时，即 SSL 握手期间对证书进行询问和验证时。如果握手失败，将不会进行连接，因为客户端和/或服务器决定不信任对方。在某些情况下，连接建立之后也可能会发生握手失败，原因如下。

- 加密或解密算法不正确。
- 缓冲区格式或大小不正确。
- （最糟糕的情况）是收到伪造消息。

29.3 检查安全套接字层故障

29.3.1 使用 SSL 调试进行探查

具体步骤如下。

- 服务器启动时使用下列 Java 标志收集 SSL 诊断数据：

```
-Dweblogic.security.SSL.verbose=true
-Dssl.debug=true
-Dweblogic.StdoutDebugEnabled=true
```

- 要做进一步（更详细）记录，可使用：

```
-Dweblogic.security.SSL.debugEaten=true
```

- SSL 调试提供有关何时产生 SSL 警报的详细信息。
- 要进行全面探查，您可能需要在 SSL 握手两端都启用调试功能。
- 可以在 SSL 调试输出中见到以下常见的警报类型：① 证书无效或过期失效；② 连接问题，例如意外消息、未知 MAC 地址；③ 加密问题，例如解密失败；④ 其他类型的问题，例如内部错误。
- 视发生警报的类型，故障排除步骤可能会有差异。

29.3.2　使用 SSL 调试输出

使用 SSL 调试输出来辨别与下列问题有关的常见警报类型。

- 证书和证书链问题，例如：certificate_expired 或 certificate_revoked，bad_certificate 或 unsupported_certificate, unknown_ca。
- 握手失败，例如：① handshake_failure；② protocol_version。
- 已建立连接的错误，例如：① close_notify；② unexpected_message 或 bad_record_mac。

29.4　检查和诊断 SSL 问题

29.4.1　SSL 证书问题及解决办法

- 如果出现证书错误，请检查：① 双方的 SSL 调试输出；② 每个证书的内容，使用以下命令，即 openssl <cmd> -in <certFile> -text，其中<cmd> 是证书的格式，例如 x509, rsa。
- 如果证书中存在错误，可能需要重新签发证书。
- 您可以采取临时措施生成一个临时（演示）证书，在获得正式证书之前暂时使用该证书。

29.4.2　SSL 证书链问题及解决办法

如果出现证书链错误，则可以采取下列办法。

- 通过检查链中的每个证书来验证证书链是否正确。
- 检查是否安装并使用了必需的证书：

① 启动时（即进行任何握手之前），客户端会记录有关密钥库中受信任 CA 的信息。

② 服务器执行握手时也会记录证书信息。

- 显式定义证书所在的密钥库。

① 在服务器启动文件中添加下列项：

```
-Dweblogic.security.TrustKeyStore=CustomTrust
-Dweblogic.security.CustomTrustKeyStoreFileName=<myKS>
```

其中，<myKS>是包含证书的密钥库。

② 或添加下列项：

```
-Dweblogic.security.SSL.trustedCAKeyStore=<myKS>
```

29.4.3　SSL 握手问题及解决办法

客户端与服务器初次握手时可能会发生问题。

① WLS（以客户端身份）连接到远程 Microsoft 服务器时，有时会发生该问题。

② 该问题可能涉及密码套件密钥大小，例如，服务器要求使用 128 位密钥。

如果密码套件密钥大小是问题所在，可能需要增加客户端密钥大小。

29.4.4　SSL 警报问题及解决办法

SSL 连接警报及解决方法如下。

- bad_mac_record:① 如果收到的记录的 MAC 地址不正确，将返回此警报；② 检查所涉及的服务器/客户端的 MAC 地址（此步骤是有效的解决方法步骤）；③ 可能表示收到伪造消息。
- unexpected_message:① 收到不适当的消息；② 正常实现的通信过程中不可能出现该警报；③ 可能表示收到伪造消息。

SSL 主机名验证警报如下。

- 如果主机名验证失败，则表示：① 客户端 URL 中使用的主机名与服务器证书中包含的主机名不匹配；② 该证书可能是为另一台计算机（例如，预生产计算机或已停用计算机）生成的。

可行的解决办法如下。

① 服务器启动时使用 Java 标志禁用主机名验证：

```
-Dweblogic.security.SSL.ignoreHostnameVerification=true
```

② 不过，禁用主机名验证会使您更容易遭受中间人攻击（Man-in-the-Middle Attack）。

③ 更好的解决办法是使用正确的主机名重新签发证书。

SSL 关闭通知警报如下。

- close_notify:① 通知接收者发送者将不再通过此连接发送任何消息，且在关闭警报后收到的任何数据都将被忽略；② 以正常方式关闭连接时，就会产生这个警告级别的警报；③ 不过，如果没有使用正确的 close_notify 消息终止连接，即双方都必须发送 close_notify，则会话将无法恢复。

解决办法：如果未识别到 close_notify，可能需要调试代码以查明应在何处处理以及如何处理它。

29.5　故障排除检查清单

故障的排除检查的主要步骤如下。

按本模式中所述启用 SSL 诊断功能。

- 使用收集到的 SSL 调试输出确定存在什么类型的问题，例如：① 证书或证书链问题；② 主机名验证问题；③ 握手问题；④ 连接建立后的问题。

探查具体的故障原因。

- 其中：① 如果存在与证书有关的问题，请分析证书和/或证书链，根据需要纠正问题；② 如果是主机名验证失败，请禁用验证或使用正确的主机名重新签发证书；③ 如果发生其他早期的握手问题，请务必检查密码套件；④ 如果会话关闭通知未得到正确处理，可能需要调试代码本身。

第 30 章　域信任问题故障

30.1　定位域信任问题故障

30.1.1　基本概念回顾

在 WebLogic Server 域之间建立信任的目的：使一个域中的 Principal（用户）可作为另一个域中的 Principal（用户）而被接受，也就是说，一位用户登录到一个域后，将允许其对另一个域进行调用。

信任建立的要求：一个域的域 Credential 属性必须与另一个域的域 Credential 匹配。

WebLogic Server 域 Credential：默认情况下在第一次启动域时随机生成，因此，默认情况下每个域的域 Credential 都不相同，要在两个域之间建立信任，必须将其显式设置为相同的值。示例如图 30-1 所示。

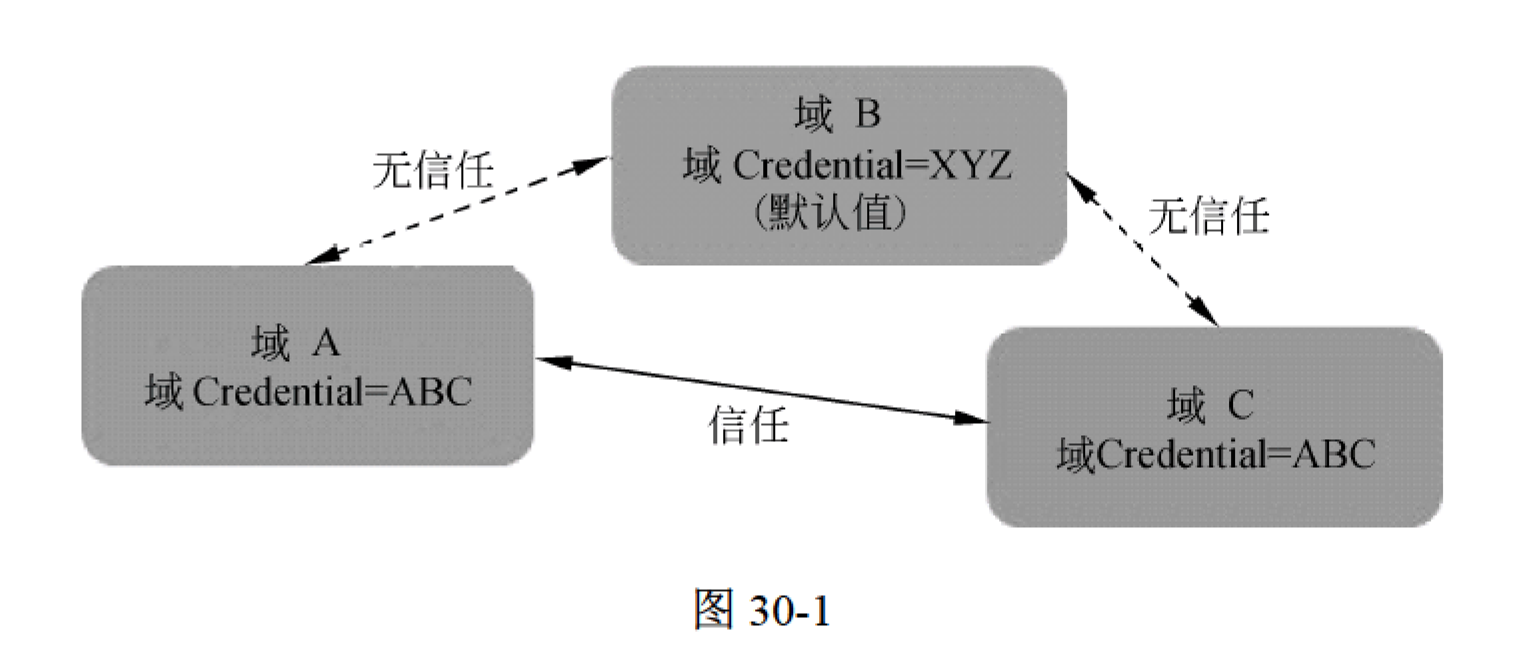


图 30-1

30.1.2　域信任考虑事项

应考虑以下域信任因素。

（1）在本地域的身份验证数据库中没有定义远程用户时，将会发生授权问题。

（2）一个域的某一组成员资格的已验证身份的用户将会继承所有受信任域中的相同组成员资格。

（3）如果域 2 既信任域 1 又信任域 3，则域 1 和域 3 之间将隐式建立相互信任关系，并因此允许进行域间用户访问。

（4）如果在一个域中扩展了 WLSUser Principal 类和 WLSGroup Principal 类，则在彼此信任的所有域中的 CLASSPATH 中也需要有这些自定义类。

30.1.3　域信任故障症状

出现域信任问题时，可能的故障症状包括两个 WebLogic Server 域之间的双向通信（例如，JNDI 查找）失败；抛出安全异常（两端）；服务器日志中出现错误消息（两端）；重新启动的管理服务器无法再与已发现的（已在运行的）被管服务器连接。

如果信任失败，WLS 域服务器将会记录输出消息：

示例 30-1：

```
java.lang.SecurityException: Authentication for user system denied in realm
wl realm
Start server side stack trace:
java.lang.SecurityException: Authentication for user system denied in realm
 wl_realm
at weblogic.security.acl.Realm.authenticate (Realm.java:212)
at weblogic.security.acl.Realm.getAuthenticatedName (Realm.java:233)
at weblogic.security.acl.internal.Security.authenticate (Security.java:171)
at weblogic.security.acl.internal.Security.verify (Security.java:95)
at weblogic.rmi.internal.BasicServerRef.handleRequest (BasicServerRef.java:292)
at weblogic.rmi.internal.BasicExecuteRequest.execute (BasicExecuteRequest.
java:22)
at weblogic.kernel.ExecuteThread.execute (ExecuteThread.java:140)
at weblogic.kernel.ExecuteThread.run (ExecuteThread.java:121)
End server side stack trace
at weblogic.rjvm.BasicOutboundRequest.sendReceive (BasicOutboundRequest.
java:108)
at weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke (ReplicaAwareRemote-
Ref.java:284)
at weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke (ReplicaAwareRemote
Ref.java:244)
at weblogic.jndi.internal.ServerNamingNode_811_WLStub.lookup (Unknown Source)
at weblogic.jndi.internal.WLContextImpl.lookup (WLContextImpl.java:338)
at ...
```

WLS81 客户端输出示例如下。

示例 30-2：

```
java.lang.SecurityException:[Security:090398]Invalid Subject:principals=
[weblogic, Administrators]
at weblogic.rjvm.BasicOutboundRequest.sendReceive (BasicOutboundRequest.
java:108)
at weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke (ReplicaAwareRemote
Ref.java:284)
at weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke (ReplicaAwareRemote
Ref.java:244)
at weblogic.jndi.internal.ServerNamingNode_812_WLStub.lookup (Unknown Source)
```

```
at weblogic.jndi.internal.WLContextImpl.lookup (WLContextImpl.java:343)
at weblogic.jndi.internal.WLContextImpl.lookup (WLContextImpl.java:336)
at javax.naming.InitialContext.lookup (InitialContext.java:347)
at bea.SourceServlet.m1 (SourceServlet.java:50)
at bea.SourceServlet.doGet (SourceServlet.java:26)
at javax.servlet.http.HttpServlet.service (HttpServlet.java:740)
at javax.servlet.http.HttpServlet.service (HttpServlet.java:853)
at weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction.run
 (ServletStubImpl.java:971)
at weblogic.security.acl.internal.AuthenticatedSubject.doAs (Authentica
tedSubject.java:317)
at weblogic.security.service.SecurityManager.runAs (SecurityManager.java:118)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet (ServletStub-
Impl.java:400)
at ...
```

在一个域内建立信任，管理服务器将在查找被管服务器的过程中记录消息。
示例 30-3：

```
Mar 31, 2009 4:41:56 PM EST> <Error> <Management> <BEA-141135> <The managed
server discovery service could not be started on the admin server.weblogic.
management.
NoAccessRuntimeException: Access not allowed for subject: principals=[],
on ResourceType: ServerRuntime Action: execute, Target: reconnectToAdminServer
at weblogic.rjvm.BasicOutboundRequest.sendReceive (BasicOutboundRequest.
java:108)
at weblogic.rmi.internal.BasicRemoteRef.invoke (BasicRemoteRef.java:138)
at weblogic.management.internal.RemoteMBeanServerImpl 812 WLStub.invoke
 (Unknown Source)
at weblogic.management.internal.MBeanProxy.invoke (MBeanProxy.java:946)
at weblogic.management.internal.MBeanProxy.invokeForCachingStub (MBean
Proxy.java:481)
at weblogic.management.runtime.ServerRuntimeMBean_Stub.reconnectToAdmin-
Server  (ServerRuntimeMBean_Stub.java:1359)
at weblogic.management.ManagedServerLocator.discoverManagedServer  (Man-
agedServerLocator.java:260)
at weblogic.management.ManagedServerLocator.discoverAllKnownServers  (Ma-
nagedServerLocator.java:130)
...Caused by: weblogic.management.NoAccessRuntimeException: Access not
allowed forsubject: principals=[], on ResourceType: ServerRuntime Action:
execute, Target: reconnectToAdminServer
```

被管服务器也将记录消息。
示例 30-4：

```
<Mar 31, 2009 4:41:56 PM EST> <Error> <Security> <BEA-090513> <ServerIdentity
failed validation, downgrading to anonymous.>
<Mar 31, 2009 4:41:56 PM EST> <Warning> <RMI> <BEA-080003> <RuntimeException
thrown by rmi server: weblogic.management.internal.RemoteMBeanServerImpl.invoke
(Ljavax.management.ObjectName;Ljava.lang.String;
[Ljava.lang.Object;[Ljava.lang.String;)
weblogic.management.NoAccessRuntimeException: Access not allowed for subj-
ect: principals=[], on ResourceType: ServerRuntime Action: execute, Target:
reconnectToAdminServer.
weblogic.management.NoAccessRuntimeException: Access not allowed for subje
ct: principals=[], on ResourceType: ServerRuntime Action: execute, Target:
reconectToAdminServer
at weblogic.management.internal.SecurityHelper$IsAccessAllowedPrivilege-
Action.wlsRun(SecurityHelper.java:564)
 at weblogic.management.internal.SecurityHelper$IsAccessAllowedPrivileg-
eAction.run(SecurityHelper.java:456)
at weblogic.security.acl.internal.AuthenticatedSubject.doAs(Authenticat-
edSubject.java:317)
 at weblogic.security.service.SecurityManager.runAs(SecurityManager.java:118)
...
```

30.1.4　设置域 Credential

1．在 WLS10.3 下配置 Credential Mappings

（1）登录管理控制台，选择左侧面板下方的 Security Realms 选项之后单击您想设定的 realm 的名字，例如 myrealm。

（2）选择 Credential Mappings 选项卡，并在该选项卡下单击 Default 按钮，如图 30-2 所示。

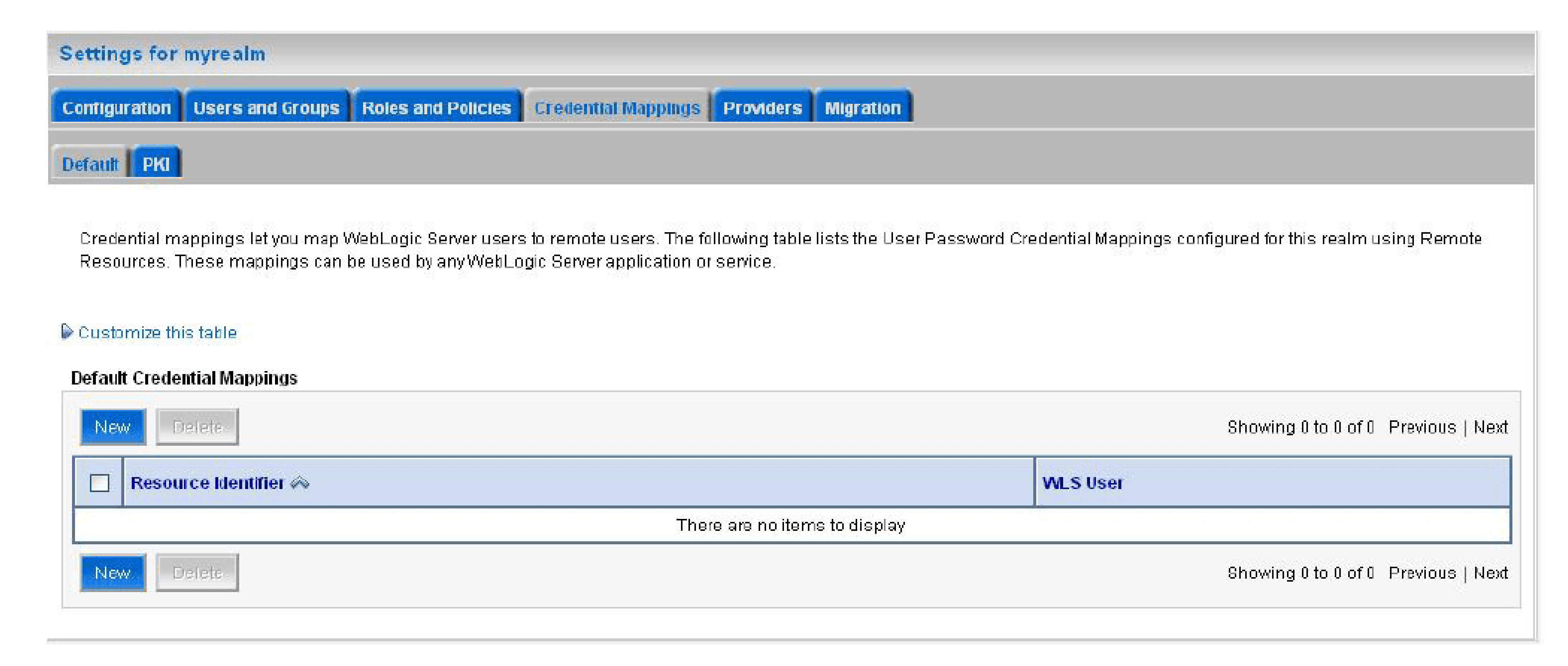


图 30-2

（3）单击 New 按钮创建远程资源，如果不使用跨 Domain 协议，请使 Use cross-domain protocol 复选框不被选中，如图 30-3 所示。

Create a New Security Credential Mapping

Back | Next | Finish | Cancel

Creating the Remote Resource for the Security Credential Mapping

Use one or more of the attributes on this page to identify the remote resource for this Credential Mapping.

* Indicates required fields

Would you like to use the cross-domain protocol as the protocol for the remote resource?

☐ Use cross-domain protocol

When not using the cross-domain protocol, remote resources are identified by the protocol, network address, path, and method that we will use in communicating with the resource. How would you like to identify the remote resource?

Protocol:

Remote Host:

Remote Port:

Path:

Method:

When using the cross-domain protocol, remote resources are identified by the name of the remote domain. How would you like to identify the remote resource?

* Remote Domain:

图 30-3

（4）单击 Next 按钮进行下一步操作，如图 30-4 所示。

Create a New Security Credential Mapping

Back | Next | Finish | Cancel

Create a New Security Credential Map Entry

Credential mappings let you map WebLogic Server users to remote users. Use this page to map a local user to a remote username and password to be used to access a remote resource.

* Indicates required fields

Specify a local user

* Local User:

Specify a remote user

* Remote User:

Specify a password for the remote user

* Remote Password:

* Confirm Password:

Back | Next | Finish | Cancel

图 30-4

（5）填写完毕后单击 Finish 按钮。

2．在 WLS10.3 下配置 Credential Mapping Providers

（1）登录管理控制台，选择左侧面板下方的 Security Realms 选项，单击您想设定的 realm 的名字，如 myrealm。

（2）选择 Providers 选项卡下的 Credential Mapping 子选项卡，如图 30-5 所示。

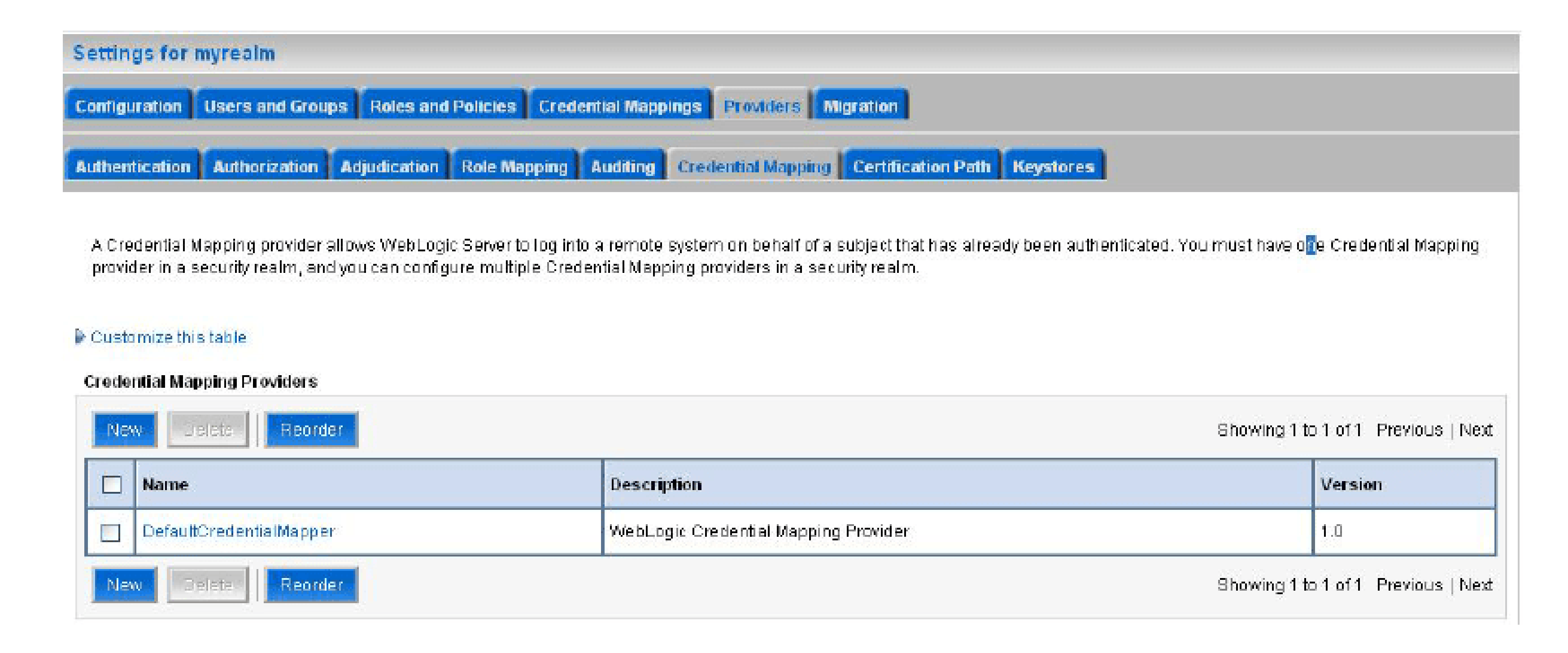


图 30-5

（3）单击 New 按钮，在 Name 文本框中输入您想为 Credential Mapping Providers 取的名字，在 Type 下拉列表框内选择 Credential Mapping Providers 的类型如图 30-6 所示。

Create a New Credential Mapping Provider

OK | Cancel

Create a new Credential Mapping Provider

The following properties will be used to identify your new Credential Mapping Provider.

* Indicates required fields

The name of the Credential Mapping Provider.

*Name: myProvider

This is the type of credential mapping provider you wish to create.

Type: SAML2CredentialMapper

OK | Cancel

图 30-6

（4）单击 OK 按钮选择刚刚创建的 Credential Mapping Providers 的名字，然后在 configuration→Provider Specific 选项卡中填写具体内容，如图 30-7 所示。

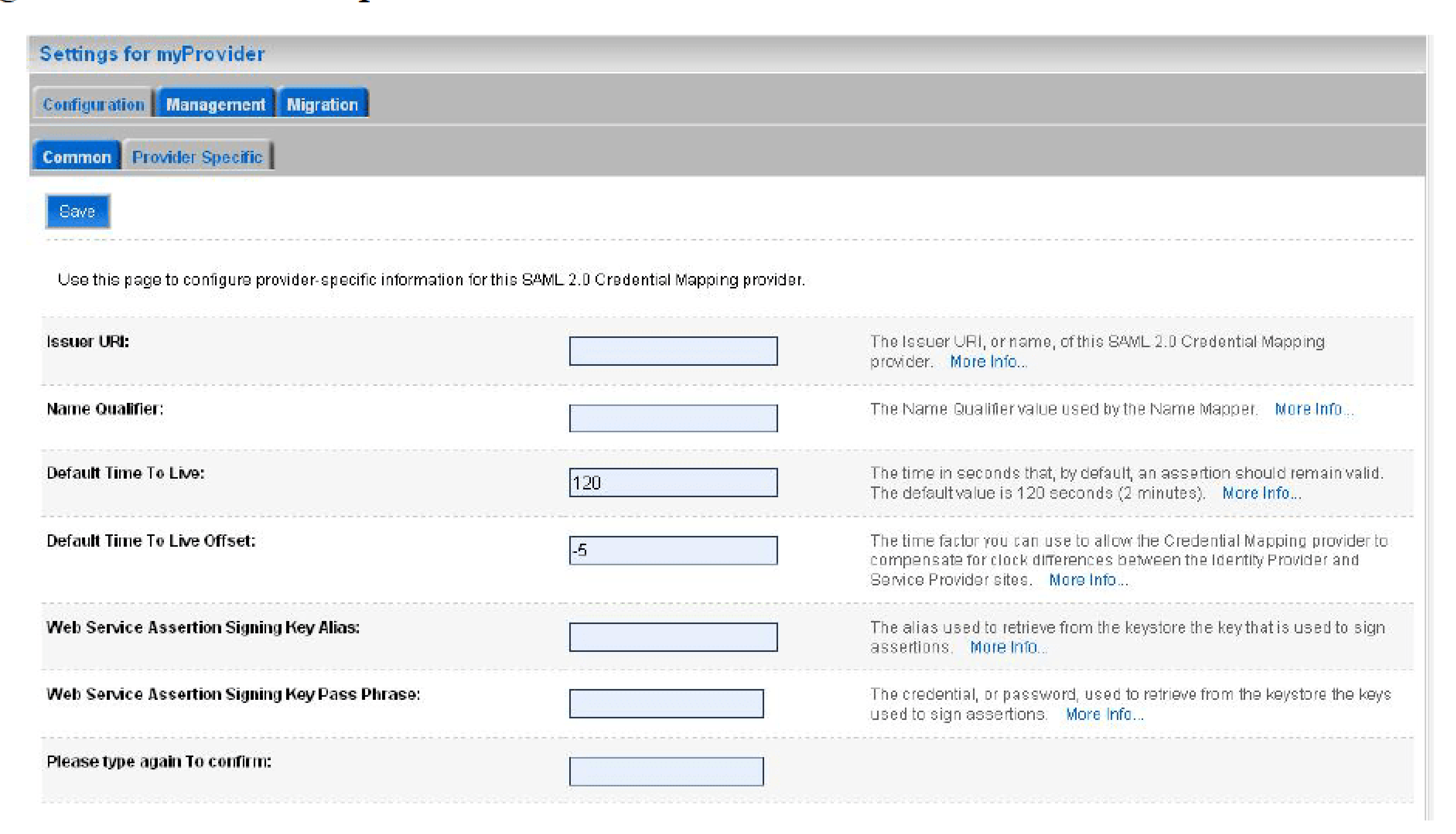


图 30-7

（5）单击“保存”按钮保存刚才所做的设置。

30.2 故障排除检查清单

故障排除策略如下。

（1）收集所有信任问题的诊断数据，例如，哪些域之间需要互操作，在哪里无法建立信任，以及所使用的 WLS 版本。

（2）设置适当的域 Credential（或系统用户密码）值，以建立信任。

（3）重新启动域的管理服务器和被管服务器，以确保域内的信任得以保留。

第 31 章　LDAP 问题故障

31.1　什么是 LDAP

轻量型目录访问协议（Lightweight Directory Access Protocol, LDAP）：是用于访问信息目录的一组协议；提供分层查找服务，包括高级搜索功能；已由许多供应商实现从 X.500 派生而来；可通过 SSL 加密。



X.500 是一个将局部名录服务连接起来，构成全球分布式的名录服务系统的协议。

31.2　WebLogic Server 对 LDAP 的支持

WLS 提供了两种 LDAP 应用方案。

31.2.1　内嵌 LDAP

对于 WLS 7.x 和 WLS 8.x，默认情况下作为管理服务器的一部分提供内嵌 LDAP 服务器（WLS 9.x 和 10.x 也是如此，图 31-1 来自 WLS10.3/11g）。

执行 Domain→Security→Embedded LDAP 命令配置内嵌的 LDAP 服务器。

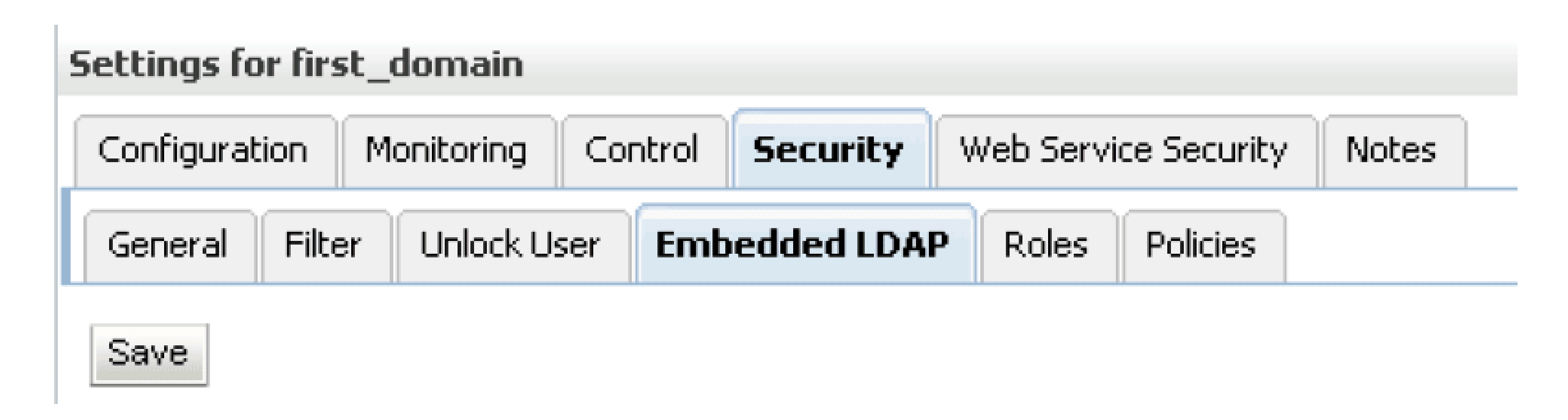


图 31-1

31.2.2　外部 LDAP

WebLogic 也支持连接到外部的 LDAP 目录服务器，该方案通常需要设置下列参数。

❑ LDAP 服务器主机和端口。

- 用于连接和进行搜索的 Principal/密码
- 用户基准 DN：在进行用户搜索时用做基准。
- 用户过滤器：在进行用户搜索时用做过滤器。
- 组基准 DN：在进行组搜索时用做基准。
- 组过滤器：在进行组搜索时用做过滤器。
- 组成员资格过滤器：在执行搜索以查找用户所属的组时用做过滤器。

以下是外部的 LDAP 目录服务器（OpenLdap）。

（1）下载并安装 OpenLdap 之后在安装路径下找到 slapd.conf，将以下行修改为自己的域名，如图 31-2 所示。

```
suffix "dc=jiexoo,dc=com"
rootdn "cn=Manager,dc=jiexoo,dc=com"
```

图 31-2

（2）另外在 include ./schema/core.schema 下面加上以下两行（图 31-3）。

```
include    ./schema/core.schema
include   ./schema/cosine.schema
include   ./schema/inetorgperson.schema
```

图 31-3

（3）cmd 进入命令行找到 OpenLdap 安装路径，运行命令/.sldapd –d 启动 OpenLdap，在出现 slapd starting 时启动成功。

（4）新建一个 LDIF（LDAP Data Interchanged Format）文件（文本文件），如图 31-4 所示。

```
dc:dc=jiexoo,dc=69fanli,dc=sg
dc:jiexoo
objectClass:top
objectClass:domain

dn:ou=people,dc=jiexoo,dc=69fanli,dc=sg
ou:people
objectClass:top
objectClass:organizationalUnit

dn:cn=tk,ou=people,dc=jiexoo,dc=69fanli,dc=sg
objectClass:person
cn:tk
sn:Tan kuan
userPassword:12345678

dn:ou=groups,dc=jiexoo,dc=69fanli,dc=sg
ou:groups
objectClass:top
objectClass:organizationalUnit

dn:cn=groupA,ou=groups,dc=jiexoo,dc=69fanli,dc=sg
cn:groupA
objectClass:top
objectClass:groupOfNames
memeber:cn=tk,ou=people,dc=jiexoo,dc=69fanli,dc=sg
```

图 31-4

执行命令导入组信息，如图 31-5 所示。

```
Ldapadd -x - D "cn=Manager,dc=jiexoo,dc=69fanli,dc=sg"-W-f test.ldif
```

```
F:\studyMiddleWare>cd OpenLDAP

F:\studyMiddleWare\OpenLDAP>ldapadd -x -D "cn=Manager,dc=jiexoo,dc=69fanli,dc=s
" -W -f test.ldif
Enter LDAP Password: ldap_bind: Invalid credentials (49)

F:\studyMiddleWare\OpenLDAP>_
```

图 31-5

（5）在 WebLogic Server Console 中执行 Security Realms→realm（默认）→Providers→New→设置 Authentication Provider 命令，如图 31-6 和图 31-7 所示。

Create a new Authentication Provider

The following properties will be used to identify your new Authentication Provider.

* Indicates required fields

The name of the authentication provider.

***Name:** joke

This is the type of authentication provider you wish to create.

Type: OpenLDAPAuthenticator

OK　Cancel

图 31-6

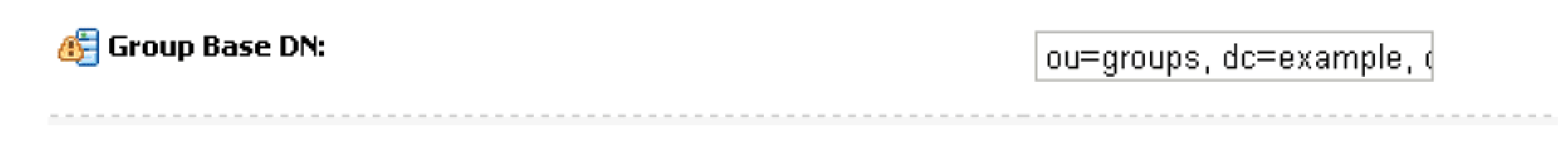


图 31-7

配置新建的 joke，如图 31-8 至图 31-10 所示。

Principal: :=jiexoo,dc=69fanli,dc=sg

Credential: ●●●●●●●●●

Confirm Credential: ●●●●●●●●●

图 31-8

User Base DN: c=jiexoo,dc=69fanli,dc=sg

图 31-9

Group Base DN: c=jiexoo,dc=69fanli,dc=sg

图 31-10

重启服务器，选择 Security Realms→myrealm→Users and Groups 选项卡，如果配置正确的话，在下面的 Users 列表中将会看到刚才在 OpenLDAP 中创建的用户 rush（图 31-11）。

	Name	Description	Provider
☐	OracleSystemUser	Oracle application software system user.	DefaultAuthenticator
☐	rush	This user is the default administrator.	DefaultAuthenticator

图 31-11

31.3 LDAP 身份验证和授权

为在 LDAP 中进行搜索，将使用明确定义了的搜索标准。

参数如图 31-12 所示。

参数	说明
dc	***域组件*** 值，定义搜索的根
ou	***组织单位***，用于区分用户组
cn	***公用名***，可能包含用户的全名
dn	***辨别名***，完全限定的用户标识符，例如使用 dc 和 ou 值进行限定
uid	用户的***用户 ID***

图 31-12

LDAP 搜索示例如下。

示例 31-1：

```
ldapsearch -b "dc=beasys,dc=com" uid=fred
```

进行身份验证和授权时，WLS：

❑ 使用已定义的 LDAP 服务器的 Principal 连接到 LDAP。

- 搜索用户并确认该用户存在。
- 使用提供的用户密码验证用户的身份。
- 搜索该用户归属的组。
- 比较用户的 Principal 与目标资源的授权所需的 Principal。
- 如果找到匹配项，则允许访问该资源，否则将拒绝访问该资源。

31.4　LDAP 安全性

1. 对于 WLS 6.x

- 默认情况下提供 File Realm。
- 缓存的 Realm，存储成功和不成功的 Realm 查找的结果以提高性能。
- LDAP Realm 被定义为自定义 Realm，必须使用连接、搜索等所需的数据进行配置。
- WLS 将尝试在服务器启动时连接到 LDAP，因此，如果 LDAP 不可用，服务器将不会启动。
- 对于 SP6 以上版本，如果 LDAP Realm 不可用，可以通过以下参数使用 File Realm 来启动服务器：-Dweblogic.security.RealmFailureOk=true。
- 在同一时间只能有一个安全 Realm 是活动的。

2. 对于 WLS 7.x 和 WLS 8.x

- 可以针对特定目的（例如身份验证、授权、裁决……）单独定义安全 Provider。
- 默认的安全 Realm（*myrealm*）存储在一个内嵌的 LDAP 服务器中。
- 在任何给定时间可以有多个安全 Realm 是活动的。
- WLS 在启动时不会连接到 LDAP，除非它需要验证用于启动的用户身份。

3. 对于 WLS 7.x 和 WLS 8.x（WLS 10.x 也有此功能）

- 可以配置多个 Authentication Provider。
- JAAS Control Flag 属性控制如何在登录序列中使用多个 Authentication Provider。

① REQUIRED。意味着将总是调用 Authentication Provider，并且用户必须始终通过相关的身份验证测试。

② SUFFICIENT。意味着如果用户通过了此 Provider 的身份验证，则不再执行其他 SUFFICIENT 或更低级别的 Authentication Provider。

③ REQUISITE。意味着如果用户通过了此 Provider 的身份验证，执行其他 SUFFICIENT 或更低级别的 Authentication Provider，但可能会失败。

④ OPTIONAL。意味着允许用户通过这些 Authentication Provider 的身份验证测试，不过，如果所有 Authentication Provider 都为 OPTIONAL，用户必须通过其中一个 Provider 的测试。

31.5 探查 LDAP 问题

31.5.1 故障症状

（1）服务器因内嵌的 LDAP 锁定问题而无法启动。
（2）Authentication Provider 无法连接到 LDAP。
（3）用户无法验证身份。
（4）用户可以验证身份，但 Authentication Provider 无法找到此用户所属的组。
（5）用户可以验证身份，但没有所需的角色。
（6）客户端对 LDAP 的身份验证速度太慢。

31.5.2 LDAP 工具及相关安全调试标志

使用下列各项来探查 LDAP 问题。

（1）WLS 安全性调试标志可用于启用由默认的身份验证、授权、裁决和角色映射安全 Provider 提供的调试记录功能

（2）LDAP 浏览器可用于访问内嵌 LDAP 服务器或其他 LDAP 服务器。

（3）ldapsearch 命令可以在 LDAP 服务器内搜索，例如，ldapsearch –b "dc=beasys, dc=com" uid=fred。

对于与安全有关的问题，config.xml 中的下列服务器调试标志可以提供更多信息。

对于 WLS 6.x，使用下列代码。

示例 31-2：

```
<ServerDebug DebugSecurityRealm="true" Name="myServer"/>
```

对于 WLS 7.x 和 8.x，使用下例代码。

示例 31-3：

```
<ServerDebug
DebugSecurityAdjudicator="true"    // adjudicator debug
DebugSecurityAtn="true"            // authentication debug
DebugSecurityAtz="true"            // authorisation debug
DebugSecurityRoleMap="true"        // role mapping debug
Name="myServer"/>
```

要动态启用这些标志，请使用 weblogic.Admin SET 命令，但切记要重新启动管理服务器。

各种工具配置事例如下。

iPlanet 配置示例如下。

示例 31-4：

```
<weblogic.security.providers.authentication.IPlanetAuthenticator
ControlFlag="SUFFICIENT"
Credential="{3DES}eFlqRhsYDyU5sqpnDRzCjg=="
DynamicGroupNameAttribute="cn"
DynamicGroupObjectClass="groupofURLs"
DynamicMemberURLAttribute="memberURL"
GroupBaseDN="ou=groups, dc=example,dc=com"
GroupFromNameFilter="(|(&amp;(cn=%g)(objectclass=groupofUniqueNames))
(&amp;(cn=%g)(objectclass=groupOfURLs)))"
Host="HOST IP or NAME OF LDAP"
Name="Security:Name=myrealmIPlanetAuthenticator"
Port="the port number of LDAP"
Principal="uid=admin, ou=Administrators, ou=TopologyManagement, o=Netsca-
peRoot"
Realm="Security:Name=myrealm"
StaticGroupDNsfromMemberDNFilter="(&amp;(uniquemember=%M)(objectclass=
groupofuniquenames))"
StaticGroupNameAttribute="cn"
StaticGroupObjectClass="groupofuniquenames"
StaticMemberDNAttribute="member"
UserBaseDN="ou=people,dc=example,dc=com"
UserFromNameFilter="(&amp;(uid=%u)(objectclass=person))"
UserNameAttribute="uid"
UserObjectClass="person"/>
```

Active Directory 配置示例如下。
示例 31-5：

```
<weblogic.security.providers.authentication.ActiveDirectoryAuthenticator
ControlFlag="SUFFICIENT" Credential="{3DES}96Kl0euDFQQ="
GroupBaseDN="CN=Users,DC=supportLDAP,DC=example,DC=com"
GroupFromNameFilter="(&amp;(cn=%g)(objectclass=group))"
Host=" HOST IP or NAME OF LDAP"
Name="Security:Name=myrealmActiveDirectoryAuthenticator"
Principal="CN=Administrator,CN=Users,DC=supportLDAP,DC=example,DC=com"
Realm="Security:Name=myrealm"
StaticGroupDNsfromMemberDNFilter="(&amp;(member=%M)(objectclass=group))"
StaticGroupNameAttribute="cn"
StaticGroupObjectClass="group"
StaticMemberDNAttribute="member"
UserBaseDN="CN=Users,DC=supportLDAP,DC=example,DC=com"
UserFromNameFilter="(&amp;(cn=%u)(objectclass=user))"
UserNameAttribute="cn" UserObjectClass="user"/>
```

OpenLDAP 配置示例如下。

示例 31-6：

```
<weblogic.security.providers.authentication.OpenLDAPAuthenticator
    ControlFlag="SUFFICIENT" Credential="{3DES}96Kl0euDFQQ="
    GroupBaseDN="ou=groups, dc=example, dc=com"
    GroupFromNameFilter="(&amp;(cn=%g)(objectclass=groupofnames))"
    Name="Security:Name=myrealmOpenLDAPAuthenticator"
    Principal="cn=Directory Manager,dc=example,dc=com"
    Realm="Security:Name=myrealm"
   StaticGroupDNsfromMemberDNFilter="(&amp;(member=%M)(objectclass=group-
  ofnames))"
    StaticGroupNameAttribute="cn"
    StaticGroupObjectClass="groupofnames"
    StaticMemberDNAttribute="member"
    UserBaseDN="ou=people, dc=example, dc=com"
    UserFromNameFilter="(&amp;(cn=%u)(objectclass=person))"
    UserNameAttribute="cn" UserObjectClass="person"/>
```

31.5.3 内嵌的 LDAP 问题

通常在服务器关机时解锁，如果发生异常停机（即服务器崩溃），则可能保持锁定状态。如果是这种情况，请确保没有其他进程正在访问该文件，然后删除它即可。

示例 31-7：

```
<Emergency> <WebLogicServer> <BEA-000342>
<Unable to initialize the server: weblogic.server.ServiceFailureException:
Could not obtain an exclusive lock to the embedded LDAP data files directory:
./server1/ldap/ldapfiles because another WebLogic Server is already using
this directory. Ensure that the first WebLogic Server is completely shutdown
and restart the server.>
```

31.5.4 LDAP 连接错误

如果 WLS 不能连接到 LDAP，常见错误如下。

示例 31-8：

```
LDAP error (49) - incorrect password (credentials)
LDAP error (32) - incorrect principal (user)
```

（1）如果 WLS 不能连接到 LDAP，需要检查下列设备。

① LDAP 服务器正在运行。

② 正确定义了 LDAP 服务器的主机名，且该主机名可以被 WLS 服务器计算机正确识别。

③ 使用了正确的端口号（默认是 389，对于 SSL 是 636）。

Principal（用户）配置为 LDAP 中的完整 DN。

示例 31-9：

```
Principal="uid=admin, ou=Administrators, ou=TopologyManagement, o=Netsc-
apeRoot"
```

（2）用户身份验证失败。

① 用户身份验证包括若干步骤。

a．WLS 首先连接到 LDAP

b．然后尝试根据下列各项搜索用户：

i．用户基准辨别名（Distinguished Name，DN）。

ii．在 Authentication Provider 自定义 Realm 中定义的用户过滤器。

c．一旦找到该用户，它就会尝试用所提供的密码进行身份验证。

每个步骤都可能导致失败，因此可启用安全性调试标志以获得更多的具体信息。

② 解决步骤如下。

确认每个步骤的要求或通过诊断来解决发现的特定问题。

用户不存在时的错误示例如下。

示例 31-10：

```
<SecurityDebug><returnConnection conn:netscape.ldap.LDAPConnection@e4bb3c>
javax.security.auth.login.FailedLoginException: [Security:090302]Authen-
tication Failed: User fred denied
```

用户密码不正确时的错误示例如下。

示例 31-11：

```
<SecurityDebug> <DN for user fred: uid=fred,ou=People, dc=beasys,dc=com>
```

找到了用户但在身份验证期间失败时的错误示例如下。

示例 31-12：

```
<SecurityDebug> <authenticate user:fred with DN:uid=fred,ou=People,dc=
beasys,dc=com>
<Debug> <SecurityDebug> <authentication failed 49>
```

（3）组成员资格问题。

身份验证完成后，WLS 会尝试确定该成员属于哪些组，检查用户是否为访问目标角色所需的组的成员。

未找到用户组的示例如下。

示例 31-13：

```
<SecurityDebug> <getDNForUser search("ou=people,dc=beasys,dc=com",
"(&(uid=fred)(objectclass=person))", base DN & below)>
<SecurityDebug> <DN for user fred: uid=fred,ou=People, dc=beasys,dc=com>
<SecurityDebug> <search("ou=groups,dc=beasys,dc=com","(&(uniquemember=
uid=fred,ou=People, dc=beasys,dc=com)(objectclass=groupofuniquenames))", base
DN & below)>
```

```
<SecurityDebug> <Result has more elements: false>
```

（4）角色映射问题如下。

① 确定了用户的组后，WLS 即会检查这些 Principal 是否有权访问目标资源所需的角色。

② 角色映射器和授权 Provider 将执行这些功能并做出授予或拒绝决定。

③ 如果使用了多个授权 Provider，裁决 Provider 将拥有最终决定权检查用户的 Principal（组）是否具有访问所需角色所必备的权限。

未授予对所需角色访问权限的示例如下。

示例 31-14：

```
<SecurityDebug> <Default RoleMapper getRoles () : input arguments:
    Subject: 2
    Principal = class weblogic.security.principal.WLSUserImpl ("weblogic")
    Principal = class weblogic.security.principal.WLSGroupImpl ("Admin")
    Resource: type=<svr>, application=, server=cgServer, action=boot>
    <SecurityDebug> <Default RoleMapper getRoles () : returning roles:
    Anonymous>
    <SecurityDebug> <RoleManager.getRoles Subject: Subject: 2
    Principal = class weblogic.security.principal.WLSUserImpl ("weblogic")
    Principal = class weblogic.security.principal.WLSGroupImpl ("Admin")
    Resource: <svr> type=<svr>, application=, server=cgServer, action=boot
    Anonymous roles.>
    <SecurityDebug> <Default Authorization isAccessAllowed () : input
    arguments:>
    <SecurityDebug> < Subject: 2
    Principal = class weblogic.security.principal.WLSUserImpl ("weblogic")
    Principal = class weblogic.security.principal.WLSGroupImpl ("Admin")
    <SecurityDebug> <Roles:Anonymous>
    <SecurityDebug> <Resource: type=<svr>, application=, server=cgServer,
    action=boot>
    <SecurityDebug> <Direction: ONCE>
    <SecurityDebug> <Context Handler: >
    <SecurityDebug> <null>
    <SecurityDebug> <Default Authorization isAccessAllowed () : returning
    DENY>
```

31.5.5 性能问题

（1）LDAP 中的用户/组过多时，如果从控制台访问它们：① 会使服务器挂起；② 或使服务器的速度变得非常慢。

（2）请在 config.xml 中的<Realm>标记上设置以下标志来禁用列举用户/组：EnumerationAllowed="false"。

（3）由于组成员资格搜索以递归方式进行，因此如果组的嵌套层次过多，身份验证性能可能会受影响。

（4）可设置下列参数来限制组成员资格层级的深度。

示例 31-15：

```
GroupMembershipSearching="limited"
MaxGroupMembershipSearchLevel="level"
```

31.6　故障排除检修清单

（1）如果内嵌的 LDAP 服务器无法启动，请检查是否存在文件锁定问题。

（2）确认是否可以连接到 LDAP 服务器。

（3）如果身份验证似为问题所在，请使用安全 Provider 调试标志来诊断问题。

（4）如果性能是问题所在，请考虑：① 禁用用户/组列举；②执行组成员资格搜索的深度。

（5）继续监视，看是否还会发生问题。

31.7　话题扩展

利用 LDAP 浏览器查看内嵌的 LDAP Server 中的内容。

（1）修改 Security 选项卡中内嵌的 LDAP 中的 Credential 和 Confirm Credential 值，如图 31-13 所示。



图 31-13

（2）启动 LDAP 浏览器设置相关的值（Password=Credential），如图 31-14 所示。

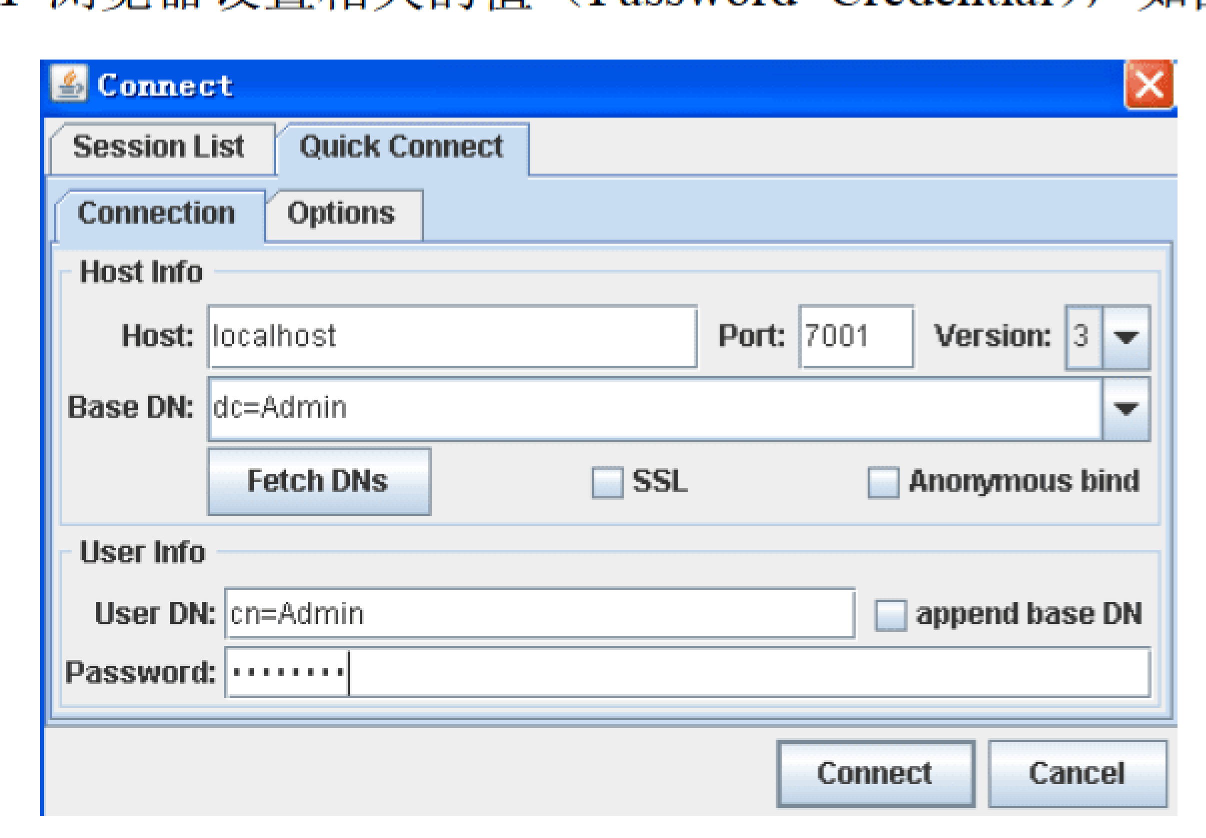


图 31-14

（3）通过 LDAP 浏览器查看并编辑 LDAP Server 里的值，如图 31-15 所示。

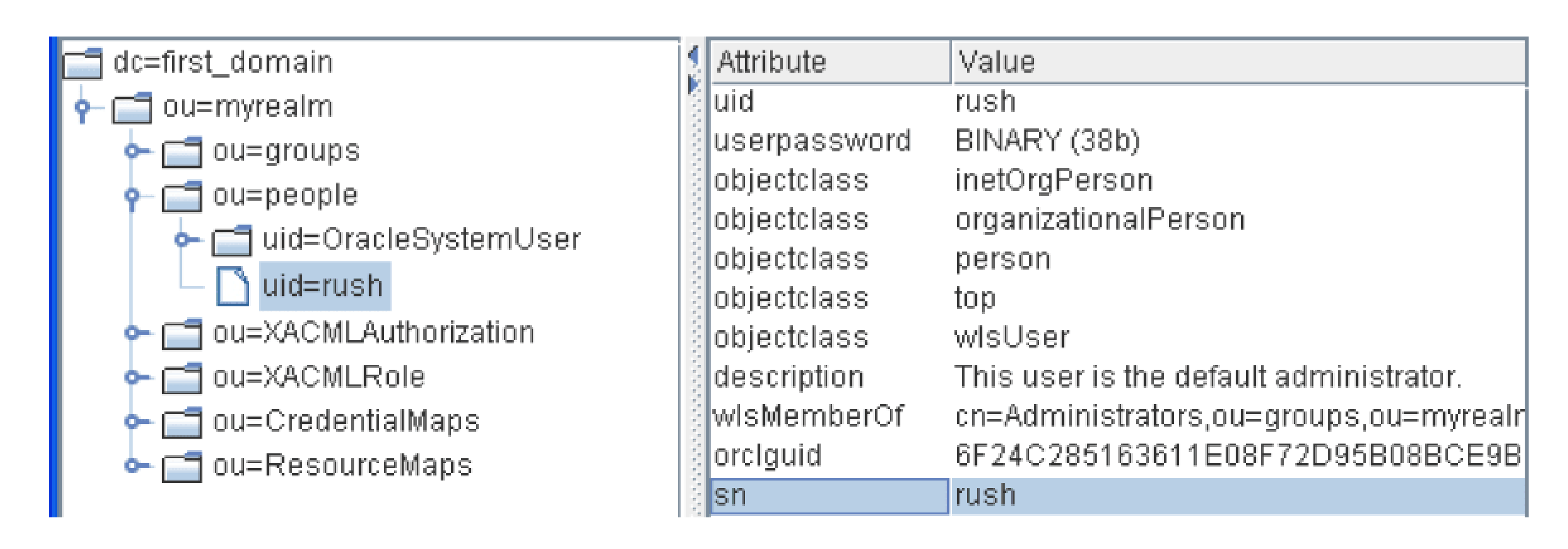


图 31-15

（4）通过访问控制文件（acls.prop），这个文件中的每一行都包含一个访问控制规则。一个访问控制规则由下面几个部分组成。

① 应用规则的 LDAP 目录的位置。

② 应用规则的位置的范围。

③ 访问权限（grant 或 deny）。

④ 许可（grant 或 deny）。

⑤ 应用规则的属性（attribute）。

⑥ 允许或拒绝访问的主题（subject）。

第 32 章　目录服务 JNDI 及其相关问题

32.1　什么是 JNDI

32.1.1　JNDI 简介

JNDI 是 Java 平台的一个标准扩展，提供了一组接口、类和关于命名空间的概念。如同其他很多 Java 技术一样，JNDI 是 provider-based 的技术，开放了一个 API 和一个服务供应接口（SPI）。这意味着任何基于名字的技术都能通过 JNDI 而提供服务，只要有对应的 JNDI SPI 实现来支持这项技术。JNDI 目前所支持的技术包括 LDAP、CORBA Common Object Service（COS）名字服务、RMI、NDS、DNS、Windows 注册表等。很多 J2EE 技术，包括 EJB 都依靠 JNDI 来组织和定位实体。

JNDI 通过绑定的概念将对象和名称联系起来。在一个文件系统中，文件名被绑定给文件。在 DNS 中，一个 IP 地址绑定一个 URL。在目录服务中，一个对象名被绑定给一个对象实体。

JNDI 中的一组绑定作为上下文来引用。每个上下文暴露的一组操作是一致的。例如，每个上下文提供了一个查找操作，返回指定名字的相应对象。每个上下文都提供了绑定和撤除绑定名字到某个对象的操作。JNDI 使用通用的方式来暴露命名空间，即使用分层上下文以及使用相同命名语法的子上下文。

命名服务提供了一种为对象命名的机制，这样您就可以在无需知道对象位置的情况下获取和使用对象。只要该对象在命名服务器上注册过，且您必须知道命名服务器的地址和该对象在命名服务器上注册的 JNDI 名，就可以找到该对象，获得其引用，从而运用它提供的服务。

JNDI 就是为 Java 中命名和目录服务定义的 Java API，是命名服务的抽象机制。可以直接通过 JNDI 来操作命名服务，而不要与底层的命名服务器交互，这大大减轻了程序员的压力。

32.1.2　应用 JNDI

1. 获得名字服务的初始环境

示例 32-1：

```
Context ctx=new InitailContext();
```

这样获得初始环境下默认的命名服务。假如您想改变提供 JNDI 服务的类（或厂商）和提供 JNDI 服务的命名服务器，可以采用以下方法。

示例 32-2：

```
Hashtable Env=new Hashtable();
Env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.enterprise.naming.Seri-
alInitContextFactory");//指定提供命名服务的类名
Env.put(Context.PROVIDER_URL,"localhost:1099");
                                          //指定提供命名服务的服务器名和端口
Context ctx=new InitialContext(env);
```

2. 对象绑定

用 bind(String name,Object o)方法，把对象 o 绑定到名字 name 上。

示例 32-3：

```
importjavax.naming.*;
public class TestJNDI{
public static void main(String[] args){
 try{
    Context ctx=new InitialContext();
    Ctx.bind("ABC","JAVA1");//把 JAVA1 字符串绑定到 ABC 上
    }
 catch(NamingException e){
    e.printStackTrace();
    }
}
}
```

如果名字已绑定或命名服务器没有启动，则会出现 NamingException 异常。

3. 重新绑定

用 ctx.rebind (String name, Object o)，如：

```
ctx.rebind("ABC","JAVA2");//现在 ABC 就绑定到 JAVA2 字符串
```

4. 解除绑定

用 ctx.unbind(String name);

不过您要确保该名字存在，否则会出现 NameNotFoundException 异常。

5. 查找已绑定的对象

用 ctx.lookup(String name);根据 name 找对象。

示例 32-4：

```
Import javax.naming.*;
```

```
Public class TestJNDI{
public static void main(String[] args){
    try{
        Context ctx=new InitialContext();
        Object o=ctx.lookup("ABC");//根据 JNDI 名查找绑定的对象
String s=(String)o;//强制转换
 }
catch(NamingException e){
        e.printStackTrace();
    }
    catch(ClassCastException e){
e.printStackTrace();
 }
    }
    }
```

6. 在 EJB 中的应用（查找 EJB HOME 对象）

示例 32-5：

```
InitialContext ic=new InitialContext();
Object o=ic.lookup("java:comp/env/ejb/Hello");  //利用 JNDI 名查找 EJB HOME
HelloHome home=(HelloHome)PortableRemoteObject.narrow(lookup,HelloHome.
class);                                          //定位 EJB//HOME 对象
Hello hello=home.create();                       //用 EJB HOME 创建 EJB 对象
```

32.2　如何使用 JNDI

1. WebLogic 中常用 JNDI 开发的几个重要的相关概念

Binding 绑定：通过名称与相应对象关联的方式称为绑定（Association of an atomic name and an object）。

Context 上下文：上下文是一组名称——对象的绑定。

Namespace：命名空间就是命名系统中的若干不同名字的集合。

Compound Name：根据命名系统规则，一系列的原子名字组成的名字。

Composite Name：不同的命名空间中的名字组成的名字。

2. 创建初始的上下文（Create an Initial Context）

示例 32-6：

```
Context ctx=new InitialContext();
```

3. 创建环境对象（Create Environment Object）

示例 32-7：

```
weblogic.jndi.Environment env=new weblogic.jndi.Environment();
env.setProviderUrl("t3://127.0.0.1:7001");      //设置服务器的 URL
env.setSecurityPrincipal("system");             //设置用户名
env.setSecurityCreentials("password");          //设置密码
Context ctx=env.getInitialContext();(获取初始上下文)
```

4. 创建子上下文

示例 32-8：

```
Context ctx=env.getInitialContext();            //获得初始上下文
Context subcontext =ctx.createSubcontext("newSubcontext");
//创建名为 newSubcontext 的子上下文
subcontext.rebind("boundobject",anobject);      //重新绑定 boundobject 对象
subcontext.close();                             //关闭子上下文
ctx.close();                                    //关闭上下文
```

5. 连接到 JNDI 初始入口的两种方法

第 1 种：
借助常量属性和 Hashtable 类。
示例 32-9：

```
Hashtable env=new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContext-
Factory");
env.put(Context.PROVIDER_URL,"t3:localhost:7001");
env.put(Context.SECURITY_PRINCIPAL,"system");
env.put(Context.SECURITY_CREDENTIALS,"password");
Context ctx=new IntialContext(env);
```

第 2 种：
借助 Properties 类。
示例 32-10：

```
Properties env=new Properties();
env.setProperty("java.naming.factory.initial","weblogic.jndi.WLInitialC-
ontextFactory");
env.setProperty("java.naming.provider.url","t3://192.2.56.102:7001");
env.setProperty("java.naming.security.principal","system");
env.setProperty("java.naming.security.credentials","password");
Context ctx=new InitialContext(env);
```

相关的环境属性说明如下。

java.naming.factory.initial：指定上下文构造工厂所使用的类名，这个类名是由服务的提供者所给定的。

java.naming.provider.url：指定服务提供者的 URL 地址。

java.naming.security.principal：用于服务验证的用户名。

java.naming.security.credentials：通过验证所需的密码。

6．从 JNDI 树中获取对象

示例 32-11：

```
import javax.naming.*;
try
{
 Context ctx=new InitialContext();                         //获取初始上下文
 Object obj;
 obj=ctx.lookup("javax.transaction.UserTransaction");  //寻找
 UserTransaction ut=( UserTransaction)obj;
 ut.begin();
 …
 ctx.close();
}catch(NamingException e){…}
```

7．绑定远程对象



远程欲被绑定的对象必须实现 Serializable 或者说必须能够被序列化。

下面给出一个绑定远程对象的操作方法。

示例 32-12：

public static Context getInitialContext() throws NamingException//次静态方法可以获得目标 WebLogic 服务器上的初始上下文。

```
{
 Enviroment env=new Environment();
 env.setProviderUrl("t3://localhost:7001");
 env.setSecurityPrincipal("system");
 env.setSecurityCredentials("weblogic");
Context context = env.getInitialContext();
return context;
}
//obtain the initial context
Context ctx=getInitialContext();
//Create a Bank object
Bank myBank = new Bank();            //创建银行对象
//Bind the object into the JNDI tree
```

```
ctx.rebind("theBank",myBank);将名称 theBank 与 myBank 对象绑定起来
ctx.close();
```

8. 去除绑定

示例 32-13：

```
public static Context getInitialContext() throws NamingException
{
    Enviroment env=new Environment();
    env.setProviderUrl("t3://localhost:7001");
    env.setSecurityPrincipal("system");
    env.setSecurityCredentials("weblogic");
    Context context = env.getInitialContext();
    return context;
}
//Obtain the intial context
Context ctx=getInitialContext();//获得初始上下文
//Unbind the object bound to the name "theBank"
ctx.unbind("theBank");//去除名称 theBank 与它之前对应对象之间的绑定
ctx.close();
```

32.3 WebLogic 中 JNDI 相关管理

32.3.1 查看 JNDI 树步骤



这里的操作以 WebLogic10.3 中的查看操作为例。

要查看 JNDI 树中的对象，可执行下列操作。

（1）在左侧窗格中展开“环境”→“服务器”子树。

（2）在服务器概要页上，单击服务器的名称，如 AdminServer。

（3）在服务器设置页上，单击“查看 JNDI 树”按钮，JNDI 树将显示在新窗口中。

（4）使用左侧面板导航到 JNDI 树：① 对于选定的节点，其内容将显示在窗口右侧的“上下文”选项卡中。② 对于选定的对象，其内容将显示在窗口右侧的“绑定”选项卡中。

32.3.2 范例

范例如图 32-1 所示。



图 32-1

32.4　WebLogic 相关 JNDI 设置问题

32.4.1　WebLogic 相关 JNDI 的设置

任何一个 Java 客户端应用程序都必须执行的第一个任务就是创建环境属性。初始化上下文工厂使用不同的属性为不同的环境定制其上下文。可以使用 Hashtable 或使用 set() 方法设置这些属性。这些名称和值一一对应的属性决定了 WLInitialContextFactory 如何创建上下文。

通常在定制初始化上下文时要使用如下属性。

（1）Context.PROVIDER_URL：确定提供命名服务的 WebLogic 服务器的 URL。默认值为 t3://localhost:7001。

（2）Context.SECURITY_PRINCIPAL：为认证的目的确定用户（即在 WebLogic 服务器安全领域定义的用户）的身份。除非此线程已经与 WebLogic 服务器用户相联系，否则此属性的默认值为 guest。

（3）Context.SECURITY_CREDENTIALS：表示登录到 WebLogic 服务器上的用户密码。

下面的代码显示如何使用这两个属性获得上下文。

示例 32-14：

```
Context ctx = null;
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContext
Factory");
```

```
    env.put(Context.PROVIDER_URL,"t3:localhost:7001");
    env.put(Context.SECURITY_PRINCIPAL,"system");
    env.put(Context.SECURITY_CREDENTIALS,"password");
    try
    {
       ctx = new InitialContext(env);
    }
    catch (NamingException e)
    {
    // ......
    }
    finally
    {
         try
         {
         ctx.close();
         }
         catch(Exception e)
         {
         // ......
         }
    }
```

32.4.2 WebLogic 中涉及 JNDI 的配置

1. JDBC 数据源（图 32-2）



图 32-2

2．JMS

JMS 模块中设置连接工厂如图 32-3 所示。

新建 JMS 系统模块资源

上一步 下一步 完成 取消

连接工厂属性

以下属性将用于标识新连接工厂。当前模块是SystemModule-0。

* 指示必需的字段

您希望如何命名您的新连接工厂?

* **名称:** ConnectionFactory-0

您希望使用什么 JNDI 名称来查找新连接工厂?

JNDI 名称:

上一步 下一步 完成 取消

图 32-3

JMS Queue 设置如图 32-4 所示。

新建 JMS 系统模块资源

上一步 下一步 完成 取消

JMS 目标属性

以下属性将用于标识新的队列。当前模块为SystemModule-0。

* 指示必需的字段

* **名称:** Queue-0

JNDI 名称:

模板: None

上一步 下一步 完成 取消

图 32-4

JMS Topic 设置如图 32-5 所示。

新建 JMS 系统模块资源

上一步 下一步 完成 取消

JMS 目标属性

以下属性将用于标识新的主题。当前模块为SystemModule-0。

* 指示必需的字段

* 名称: Topic-0

JNDI 名称:

模板: None

上一步 下一步 完成 取消

图 32-5

3. 其他

如果是 EJB 部署成功，也都会在 JNDI 树上有显示。

第 33 章　管理框架 JMX 及控制台的相关问题

33.1　JMX 简介

JMX（Java Management Extensions，Java 管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统、网络和服务管理应用。这使得为应用程序添加管理特性变得非常容易。

伸缩性的架构使得每个 JMX Agent 服务可以很容易地放入到 Agent 中；各类 JMX 的实现都提供几个核心的 Agent 服务，用户也可以自己编写服务，并且服务可以很容易地部署或取消部署。

JMX 主要提供接口，允许有不同的实现；WebLogic 的管理页面就是基于 JMX 实现的。JMX 框架如图 33-1 所示。

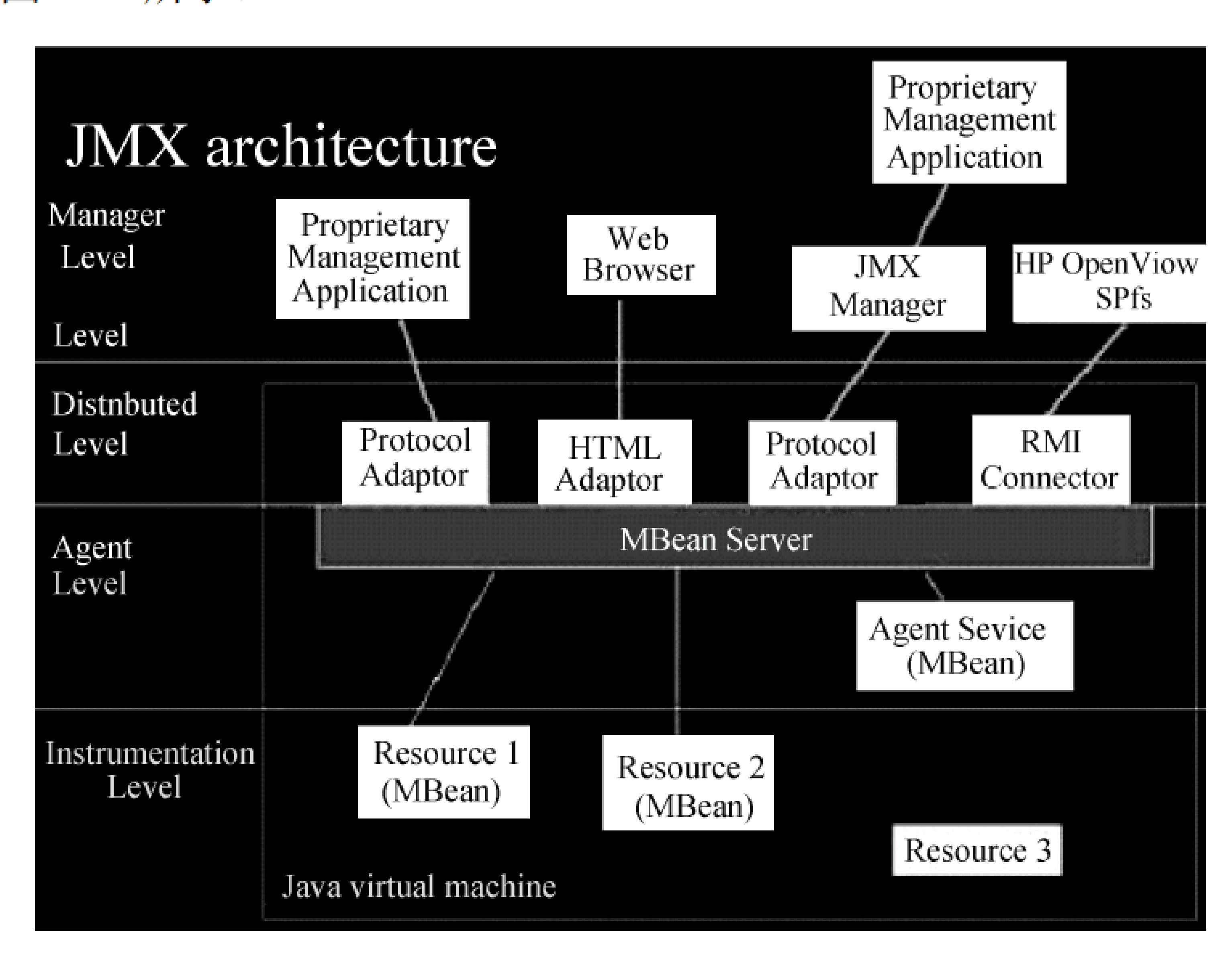


图 33-1　JMX 框架

33.2　JMX 架构中的各层及相关的组件

JMX 体系结构分为以下 4 个层次。

33.2.1 设备层

设备层（Instrumentation Level）包括下列各部分。

（1）MBeans。

（2）通知模型：Notification、NotificationListener 等类。

（3）MBean 元数据类：Attribute、Operator 等类。

该层定义了如何实现 JMX 管理资源的规范。一个 JMX 管理资源可以是一个 Java 应用、一个服务或一个设备，它们可以用 Java 开发，或者至少能用 Java 进行包装，并且能被置入 JMX 框架中，从而成为 JMX 的一个管理构件（Managed Bean，MBean）。管理构件可以是标准的，也可以是动态的，标准的管理构件遵从 JavaBeans 构件的设计模式；动态的管理构件遵从特定的接口，提供了更大的灵活性。

该层还定义了通知机制以及实现管理构件的辅助元数据类。

1. 管理构件（MBean）

在 JMX 规范中，管理构件定义如下：它是一个能代表管理资源的 Java 对象，遵从一定的设计模式，还需实现该规范定义的特定的接口。该定义保证了所有的管理构件以一种标准的方式来表示被管理资源。

管理接口就是被管理资源暴露出的一些信息，通过对这些信息的修改就能控制被管理资源。一个管理构件的管理接口包括以下内容。

（1）能被接触的属性值。

（2）能够执行的操作。

（3）能发出的通知事件。

（4）管理构件的构建器。

管理构件通过公共的方法以及遵从特定的设计模式封装了属性和操作，以便暴露给管理应用程序。例如，一个只读属性在管理构件中只有 Get 方法，既有 Get 又有 Set 方法表示是一个可读写的属性。

其余的 JMX 的构件，例如 JMX 代理提供的各种服务，也是作为一个管理构件注册到代理中才能提供相应的服务。

JMX 对管理构件的存储位置没有任何限制，管理构件可以存储在运行 JMX 代理的 Java 虚拟机的类路径的任何位置，也可以从网络上的任何位置导入。

JMX 定义了 4 种管理构件：标准、动态、开放和模型管理构件。每一种管理构件可以根据不同的环境需要进行制定。

（1）标准管理构件。

标准管理构件的设计和实现是最简单的，它们的管理接口通过方法名来描述。标准管理构件的实现依靠一组命名规则，称之为设计模式。这些命名规则定义了属性和操作。检查标准管理构件接口和应用设计模式的过程被称为内省（Introspection）。JMX 代理通过内省来查看每一个注册在 MBean 服务器上的管理构件的方法和超类，看它是否遵从一定设计模式，决定它是否代表了一个管理构件，并辨认出它的属性和操作。

（2）动态管理构件。

动态管理构件提供了更大的灵活性，它可以在运行期暴露自己的管理接口。它的实现是通过实现一个特定的接口 DynamicMBean。

JMX 代理通过 getMBeanInfo 方法来获取该动态管理构件暴露的管理接口，该方法返回的对象是 MbeanInfo 类的实例，包含了属性和操作的签名。由于该方法的调用是发生在动态管理构件向 MBean 服务器注册以后，因此管理接口是在运行期获取的。不同于标准管理构件，JMX 代理不需要通过内省机制来确定动态管理构件的管理接口。由于 DynamicMBean 的接口是不变的，因此可以屏蔽实现细节。由于这种在运行期获取管理接口的特性，动态管理构件提供了更大的灵活性。

（3）开放管理构件。

开放管理构件是一种专门化的动态管理构件，其中所有的与该管理构件相关的参数、返回类型和属性都围绕一组预定义的数据类型（String、Integer、Float 等）来建立，并且通过一组特定的接口来进行自我描述。JMX 代理通过获得一个 OpenMBeanInfo 对象来获取开放管理构件的管理接口，OpenMBeanInfo 是 MbeanInfo 的子类。

（4）模型管理构件。

模型管理构件也是一种专门化的动态管理构件。它是预制的、通用的和动态的 MBean 类，已经包含了所有必要默认行为的实现，并允许在运行时添加或覆盖需要定制的那些实现。JMX 规范规定该类必须实现为 javax.management.modelmbean. RequiredModelMBean，管理者要做的就是实例化该类，并配置该构件的默认行为并注册到 JMX 代理中，即可实现对资源的管理。JMX 代理通过获得一个 ModelMBeanInfo 对象来获取管理接口。

模型管理构件具有以下新的特点。

① 持久性。定义了持久机制，可以利用 Java 的序列化或 JDBC 来存储模型 MBean 的状态。

② 通知和日志功能。能记录每一个发出的通知，并能自动发出属性变化通知。

③ 属性值缓存。具有缓存属性值的能力。

2．通知模型

一个管理构件提供的管理接口允许代理对其管理资源进行控制和配置。然而，对管理复杂的分布式系统来说，这些接口只是提供了一部分功能。通常，管理应用程序需要对状态变化或者当特别情况发生变化时做出反映。

为此，JMX 定义了通知模型。通知模型仅仅涉及了在同一个 JMX 代理中的管理构件之间的事件传播。JMX 通知模型依靠以下几个部分。

（1）Notification：一个通用的事件类型，该类标识事件的类型，可以被直接使用，也可以根据传递的事件的需要而被扩展。

（2）NotificationListener 接口：接受通知的对象需要实现此接口。

（3）NotificationFilter 接口：作为通知过滤器的对象需要实现此接口，为通知监听者提供了一个过滤通知的过滤器。

（4）NotificationBroadcaster 接口：通知发送者需要实现此接口，该接口允许希望得到

通知的监听者注册。

发送一个通用类型的通知，任何一个监听者都会得到该通知。因此，监听者需要提供过滤器来选择所需要接受的通知。

任何类型的管理构件，标准的或动态的，都可以作为一个通知发送者，也可以作为一个通知监听者，或两者都是。

3. 辅助元数据类

辅助元数据类用来描述管理构件。辅助元数据类不仅被用来内省标准管理构件，也被动态管理构件用来进行自我描述。这些类根据属性、操作、构建器和通告描述了管理接口。JMX 代理通过这些元数据类管理所有管理构件，而不管这些管理构件的类型。

部分辅助元类如下。

（1）MBeanInfo：包含了属性、操作、构建器和通知的信息。

（2）MBeanFeatureInfo：为下面类的超类。

（3）MBeanAttributeInfo：用来描述管理构件中的属性。

（4）MBeanConstructorInfo：用来描述管理构件中的构建器。

（5）MBeanOperationInfo：用来描述管理构件中的操作。

（6）MBeanParameterInfo：用来描述管理构件操作或构建器的参数。

（7）MBeanNotificationInfo：用来描述管理构件发出的通知。

33.2.2 代理层

代理层（Agent Level）主要定义了各种服务以及通信模型。该层的核心是一个 MBean Server，所有的管理构件都需要向它注册才能被管理。注册在 MBean Server 上管理构件并不直接和远程应用程序进行通信，它们通过协议适配器和连接器进行通信。而协议适配器和连接器也以管理构件的形式向 MBean Server 注册才能提供相应的服务。

代理服务可以对注册的管理构件执行管理功能。通过引入智能管理，JMX 可以帮助人们建立强有力的管理解决方案。代理服务本身也是作为管理构件而存在的，也可以被 MBean 服务器控制。

1. JMX 规范定义的代理服务

（1）动态类装载：通过管理小程序服务可以获得并实例化新的类，还可以使位于网络上的类库本地化。

（2）监视服务：监视管理构件的属性值变化，并将这些变化通知给所有的监听者。

（3）时间服务：定时发送一个消息或作为一个调度器使用。

（4）关系服务：定义并维持管理构件之间的相互关系。

代理层的组成如图 33-2 所示。

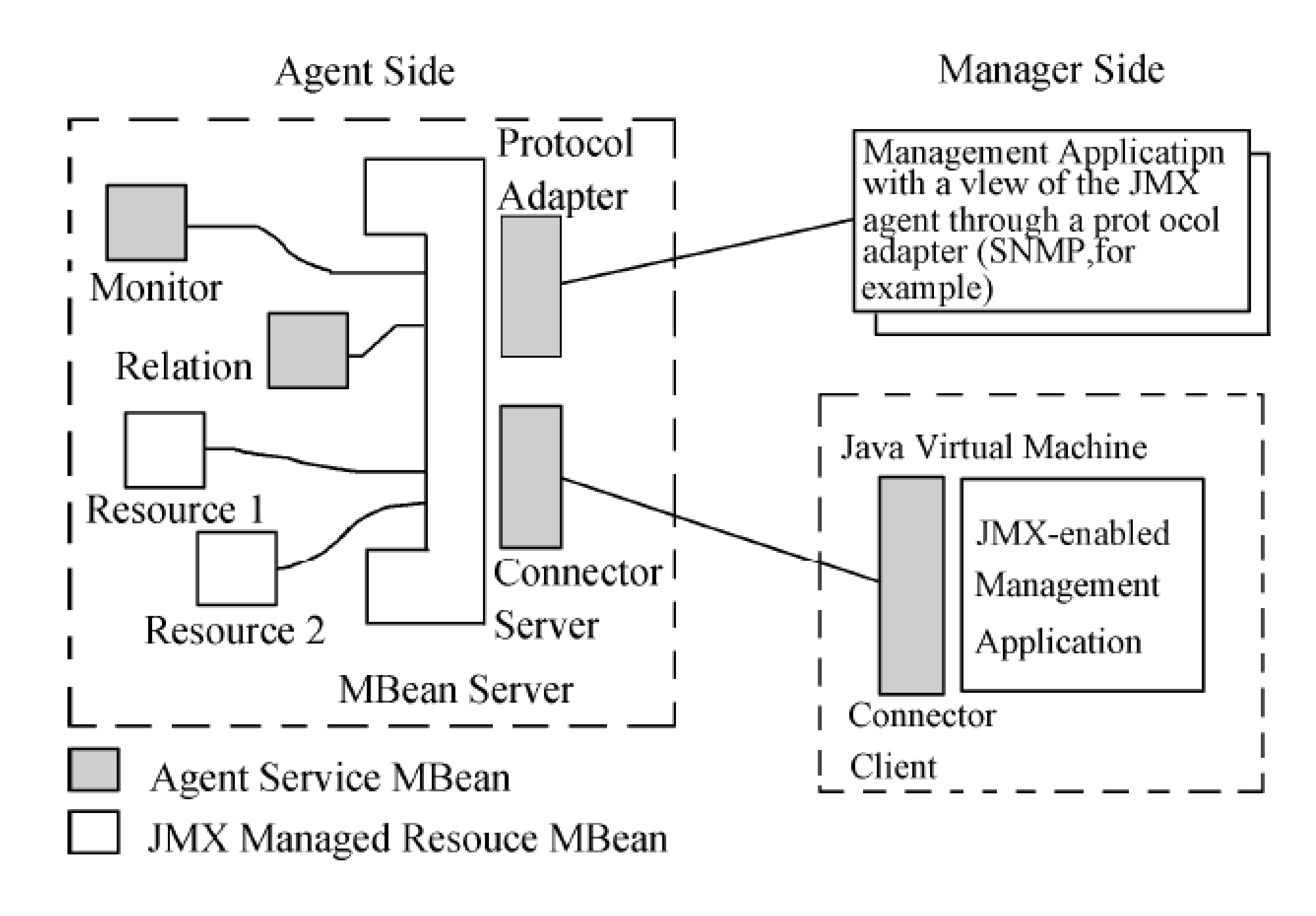


图 33-2

2．动态类装载

动态类装载是通过 m-let（management applet）服务来实现的，它可以从网络上的任何 URL 处下载并实例化管理构件，然后向 MBean 服务器注册。在一个 M-let 服务过程中，首先是下载一个 m-let 文本文件，该文件是 XML 格式的文件，文件的内容标识了管理构件的所有信息，比如构件名称、在 MBean 服务器中唯一标识该构件的对象名等。然后根据这个文件的内容，m-let 服务完成剩余的任务。图 33-3 所示为这一过程。

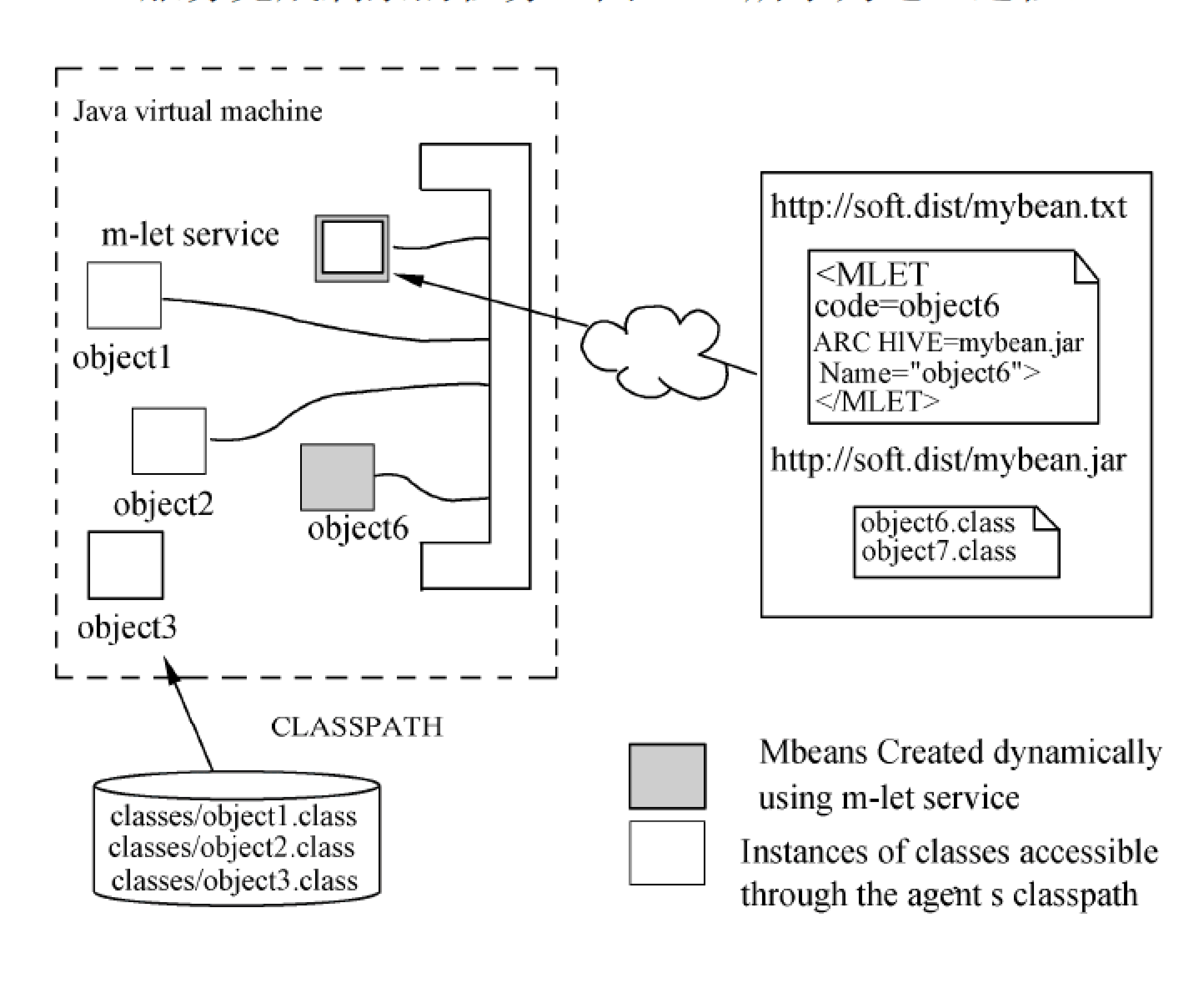


图 33-3

3．监视服务

通过使用监视服务，管理构件的属性值就会被定期监视，从而保证始终处于一个特定

的范围。当监视的属性值的变化超出了预期定义的范围，一个特定的通告就会发出。JMX 规范当前规定了 3 种监视器。

（1）计数器监视器：监视计数器类型的属性值，通常为整型，且只能按一定规律递增。

（2）度量监视器：监视度量类型的属性值，通常为实数，值能增能减。

（3）字符串监视器：监视字符串类型的属性值。

每一个监视器都是作为一个标准管理构件存在的，需要提供服务时，可以由相应的管理构件或远程管理应用程序动态创建并配置注册使用。

图 33-4 示意了计数器监视器的使用情况。

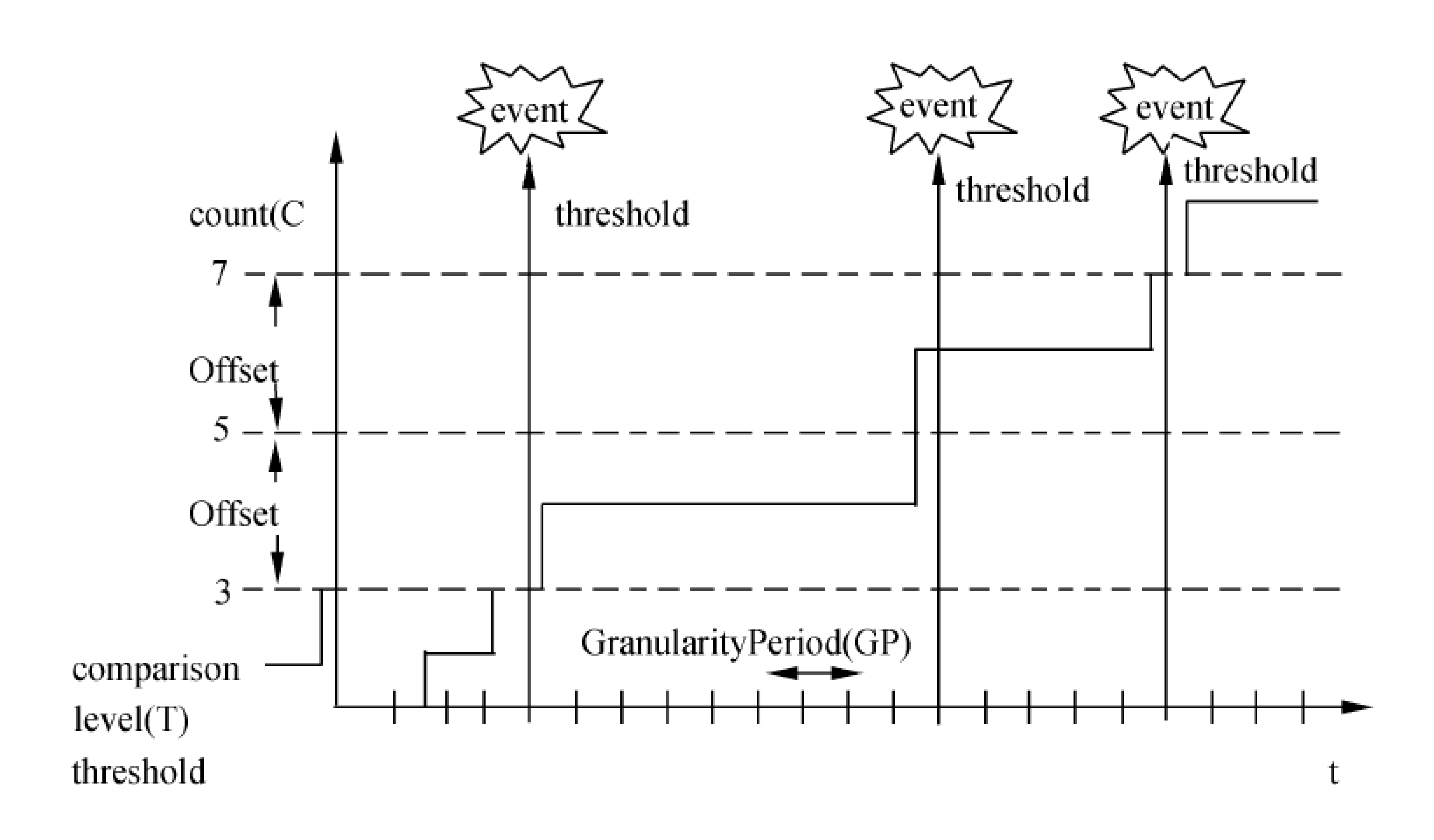


图 33-4

4．时间服务

时间服务可以在制定的时间和日期发出通告，也可以定期地、周期性地发出通告，依赖于管理应用程序的配置。时间服务也是一个管理构件，它能帮助管理应用程序建立一个可配置的备忘录，从而实现智能管理服务。

5．关系服务

JMX 规范定义了管理构件之间的关系模型。一个关系是用户定义的管理构件之间的 N 维联系。关系模型定义如下一些术语。

（1）角色：就是一个关系中的一类成员身份，它含有一个角色值。

（2）角色信息：描述一个关系中的一个角色。

（3）关系类型：由角色信息组成，作为创建和维持关系的模板。

（4）关系：管理构件之间的当前联系，且必须满足一个关系类型的要求。

（5）角色值：在一个关系中当前能满足给定角色的管理构件的列表。

（6）关系服务：是一个管理构件，能接触和维持所有关系类型和关系实例之间的一致性。

在关系服务中，管理构件之间的关系由通过关系类型确定的关系实例来维护。仅仅只

有注册到 MBean 服务器上并且能被对象名标识的管理构件才能成为一个关系的成员。关系服务从来就不直接操作它的成员——管理构件，为了方便查找，它仅仅提供了对象名。

关系服务能锁定不合理关系类型的创建，同样，不合理的关系的创建也会被锁定。角色值的修正也要遵守一致性检查。

由于关系是定义在注册的管理构件之间的联系，所以当其中的管理构件卸载时，就会更改关系。关系服务会自动更改角色值。所有对关系实例的操作比如创建、更新、删除等都会使关系服务发出通告，通告会提供有关这次操作的信息。

JMX 关系模型只能保证所有的管理构件满足它的设计角色，也就是说，不允许一个管理构件同时出现在许多关系中。

33.2.3　分布服务层

分布服务层（Distributed Service Level）主要定义了能对代理层进行操作的管理接口和构件，这样管理者就可以操作代理。然而，当前的 JMX 规范并没有给出这一层的具体规范。

33.3　一个简单的 JMX 应用

33.3.1　配置环境

开发环境：JDK5.0＋Eclipse3.2。



因为用到 jmxtools.jar 中的 HtmlAdaptorServer 类，所以将此包加入到项目库引用中。

33.3.2　一个简单的 JMX 应用的代码

（1）要管理 Hello 必须创建 MBean，代码如下。

示例 33-1：

```
package com.landingbj.copy;
public interface HelloMBean {
    public String getName();
     public void setName(String name);
     public void printHello();
     public void printHello(String whoName);

}
```

（2）Hello 是一个需要被管理的类。

示例 33-2：

```
package com.landingbj.copy;

public class Hello implements HelloMBean {
    private String name="this is my frist jmx";
    private String whoName="this is jack name";

    public String getName() {
        return name;
    }
    public void setName(String name) {

         this.name = name;
    }
    public String getWhoName() {
        return whoName;
    }
    public void setWhoName(String whoName) {
        this.whoName = whoName;
    }
    public void printHello() {
        System.out.println("Hello World, " + name);
    }

    public void printHello(String whoName) {
        System.out.println("Hello , " + getWhoName());

    }

}
```

（3）创建一个 Agent 类。

示例 33-3：

```
package com.landingbj.copy;
import java.lang.management.ManagementFactory;
import javax.management.MBeanServer;
import javax.management.ObjectName;

import com.sun.jdmk.comm.HtmlAdaptorServer;


public class HelloAgent {
    public static void main(String[] args) throws Exception {
        //MBeanServer server = MBeanServerFactory.createMBeanServer();
        MBeanServer server = ManagementFactory.getPlatformMBeanServer();
        ObjectName helloName = new ObjectName("weblogic:name=HelloWorld");
                 server.registerMBean(new Hello(), helloName);
                 ObjectName adapterName = new ObjectName(
                 "HelloAgent:name=htmladapter,port=8082");
                 HtmlAdaptorServer adapter = new HtmlAdaptorServer();
                 server.registerMBean(adapter, adapterName);
                 adapter.start();
```

```
                System.out.println("start.....");
        }
    }
```

33.3.3 说明

① 先创建了一个 MBeanServer，用来做 MBean 的容器。

② 将 Hello 这个类注入到 MBeanServer 中，注入需要创建一个 ObjectName 类。

③ 创建一个 AdaptorServer，这个类将决定 MBean 的管理界面，这里用最普通的 Html 型界面。AdaptorServer 其实也是一个 MBean。

④ weblogic：name=HelloWorld 的名字是有一定规则的，格式为："域名：name=MBean 名称"，域名和 MBean 名称都可以任意取。

33.3.4 运行 HelloAgent 测试

打开网页：http://localhost:8082/，看效果，如图 33-5 所示。



图 33-5

33.3.5 使用 JDK 的 Jconsole 来连接 Mbean

前面所有看效果都是通过 Html 网页来看的。JDK1.6 自带了一个 JMX 客户端，叫 jconsole，位于 C:\Program Files\Java\jdk1.6.0_02\bin\jconsole.exe。我们用这个客户端来连接 Mbean Server。

1. 用 RMI 方式

还是用前面的 HelloAgent 代码加上一段代码，启动一个 JMXConnectorServer 服务。示例 33-4：

```
package com.landingbj.copy.copy;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;
import com.sun.jdmk.comm.HtmlAdaptorServer;
public class HelloAgent {
    public static void main(String[] args) throws Exception {

        MBeanServer server = MBeanServerFactory.createMBeanServer();
         ObjectName helloName = new ObjectName("weblogic:name=HelloWorld");
         Hello hello = new Hello();
         server.registerMBean(hello, helloName);
        ObjectName adapterName = new ObjectName("HelloAgent:name=htmlada-
        pter,port=8082");
         HtmlAdaptorServer adapter = new HtmlAdaptorServer();
         server.registerMBean(adapter, adapterName);
        adapter.start();
         System.out.println("start.....");

         // Create an RMI connector and start it
         JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/-
         rmi://localhost:9999/server");

         //用命令 rmiregistry 启动 RMI 注册服务的

         JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnec-
         torServer(url, null, server);
        cs.start();
     System.out.println("rmi start.....");
    }
}
```

2. 运行

在 DOS 下运行：

```
rmiregistry 9999
```

然后运行 HelloAgent，启动 jconsole 并按图 33-6 所示的输入信息。

```
service:jmx:rmi:///jndi/rmi://localhost:9999/server
```

最后效果如图 33-7 所示。

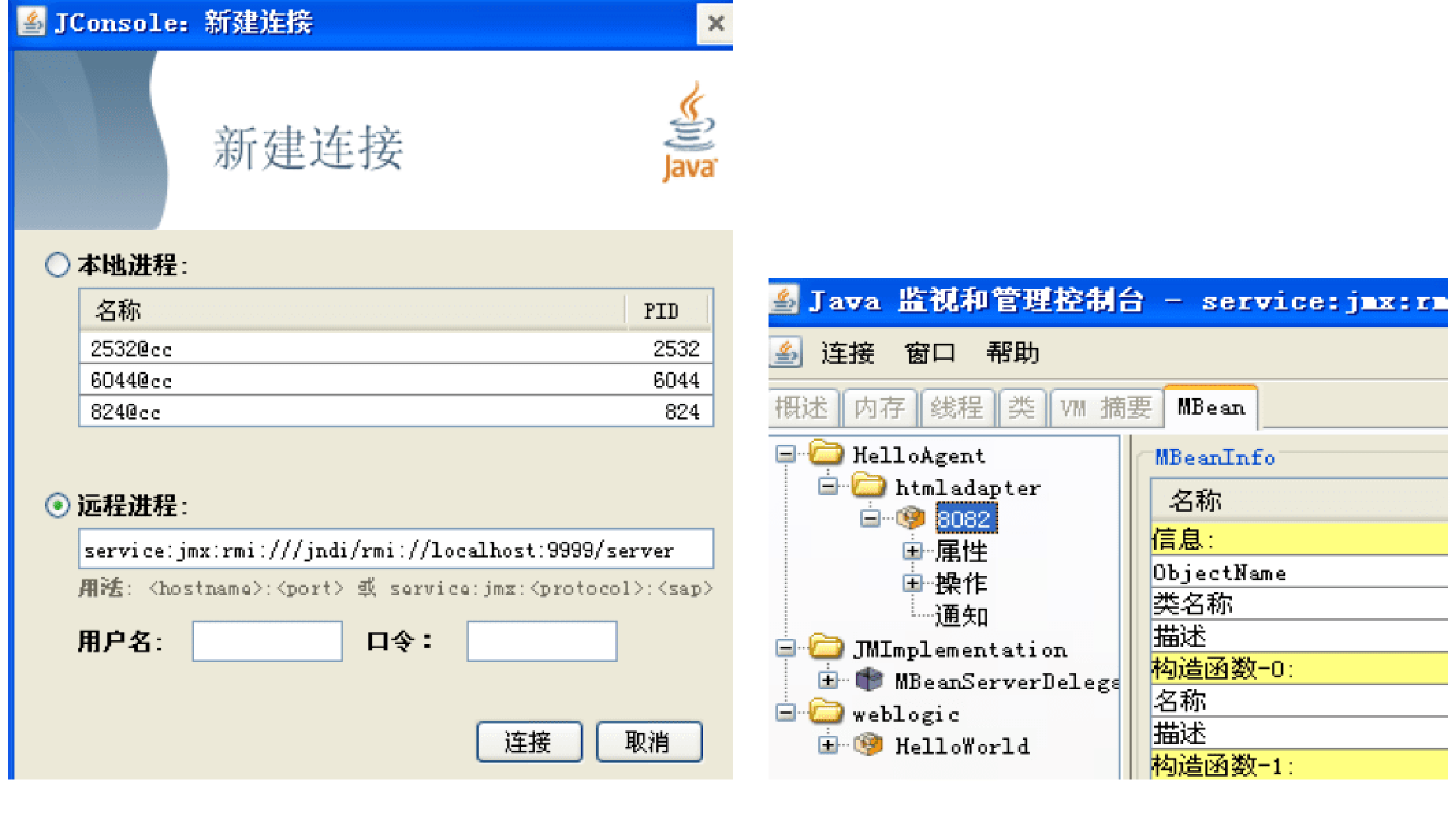


图 33-6　　　　图 33-7

33.4　WebLogic 诊断框架

33.4.1　什么是 WebLogic 诊断框架

WebLogic 诊断框架（WebLogic Diagnostic Framework，WLDF）是一种监视和诊断框架，用于定义和实现一组在 Oracle WebLogic Server® 进程中运行并参与标准服务器生命周期的服务。使用 WLDF，可以创建、收集、分析、归档和访问由运行的服务器和部署在其容器中的应用程序生成的诊断数据。通过此数据可以了解服务器的运行时的性能。

WLDF 包括多个用于收集和分析数据的组件。

（1）诊断图像捕获：从服务器创建可用于故障后分析的诊断快照。

（2）归档文件：捕获并持久保存来自服务器实例和应用程序的数据事件、日志记录和规格。

（3）测量：将诊断代码添加到 WebLogic Server 实例和运行在其上的应用程序，以执

行代码中指定位置的诊断操作。测量组件提供了将诊断上下文与请求相关联的方法，以便在请求流过系统时可对其进行跟踪。

（4）收集器：从运行时 Mbean（包括 WebLogic Server Mbean 和自定义 Mbean）捕获规格（可归档这些规格，并可随后对其进行访问，以用于查看历史数据）。

（5）观察器和通知：提供用于监视服务器和应用程序状态并根据观察器中设置的条件发送通知的方法。

（6）日志记录服务：管理用于监视服务器、子系统和应用程序事件的日志。WebLogic Server 日志记录服务是独立于 WebLogic 诊断框架的其余部分来记录的。

WLDF 提供一组标准化的应用程序编程接口（API），通过这些 API 可以对诊断数据进行动态访问和控制，以及改进的监视功能（该监视功能可以提供对服务器状态的可见性）。独立软件供应商（Independent Software Vendor，ISV）可使用这些 API 来开发用于与 WLDF 集成的自定义监视和诊断工具。

33.4.2 WLDF 诊断框架结构体系概述

1. WLDF 简介

WLDF 包含以下部分，如图 33-8 所示。

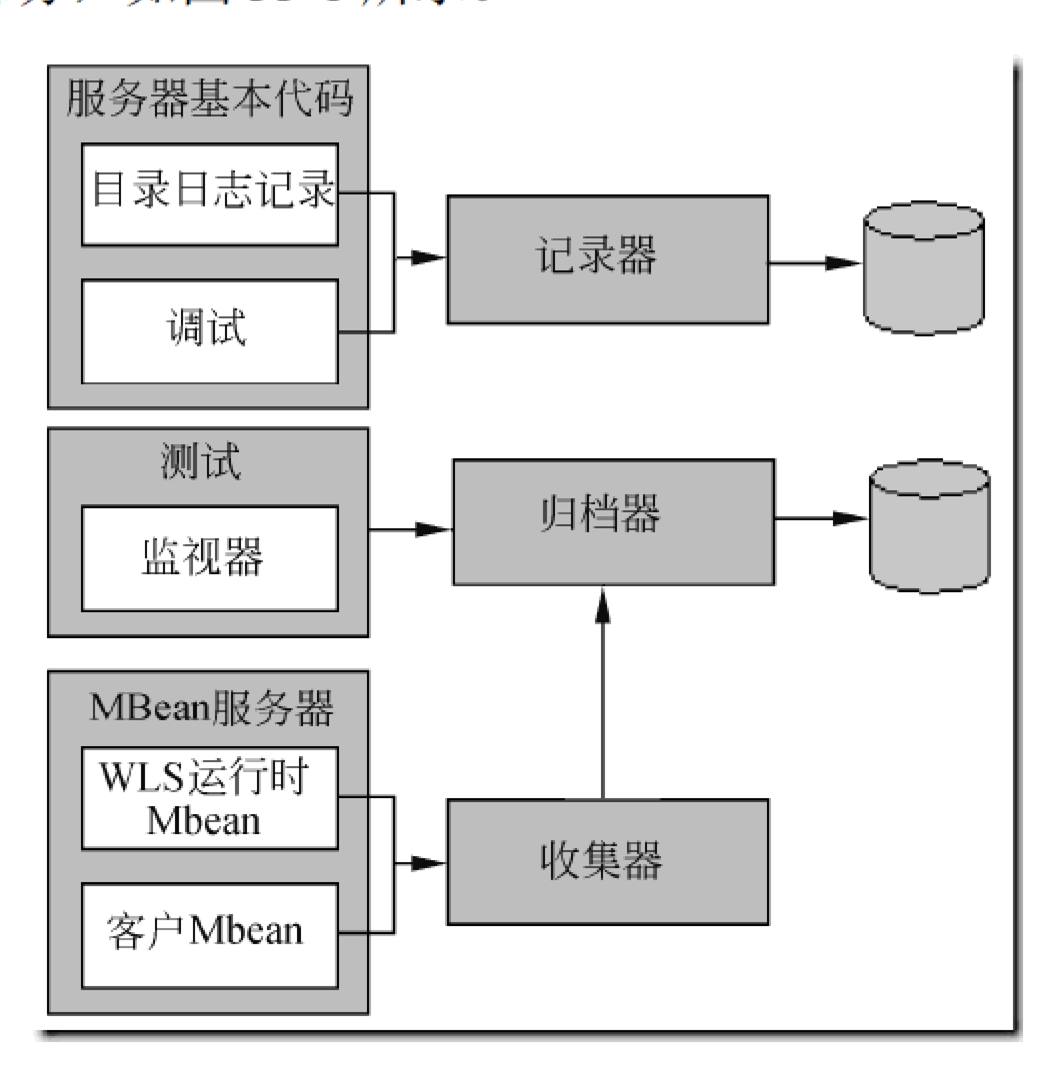


图 33-8

2. 数据创建程序分布在各 WLDF 组件之间的数据发布程序和数据提供程序

- 诊断数据是从许多来源收集的。在逻辑上，这些来源可分为两类，一类为数据提供程序（数据创建程序，会对其定期采样以收集当前值），一类为数据发布程序（数据创建程序，可同步生成事件）。数据提供程序和数据发布程序分布在各组件

之间，并且，生成的数据可由记录器和/或收集器收集。

- 服务器日志记录基础结构的调用可用做内联数据发布程序，并且生成的数据是作为事件进行收集的。（可通过目录基础结构、调试模型来调用日志记录基础结构，或直接通过记录器来调用日志记录基础结构。）
- 测量系统可创建监视器，并将其插入执行流中已定义完善的点上。这些监视器可将数据直接发布到归档文件中。

注册到 MBean 服务器中的组件也可以通过注册到收集器，使其作为数据提供程序。然后，收集到的数据会对观察器和通知系统公开（以用于自动监视），也会对归档文件公开（以便持久保存）。

3. 归档文件组件（图 33-9）

- 过去的状态通常对诊断系统中的故障十分关键。这就要求可以捕获并归档状态（创建历史归档文件）以供将来访问。在 WLDF 中，归档文件通过几个持久性组件满足了此要求。事件和收集到的规格可持久保存并可用于历史回顾。
- 传统的日志记录信息（该信息对人是可读的，并且专门包含在服务器日志中）是通过标准日志记录追加程序持久保存的。专供系统使用的新的事件数据是使用事件归档程序持久保存到事件存储中的。规格数据是使用数据归档程序持久保存到数据存储中的。
- 归档文件提供了访问接口以便访问器可公开任何持久保存的历史数据。

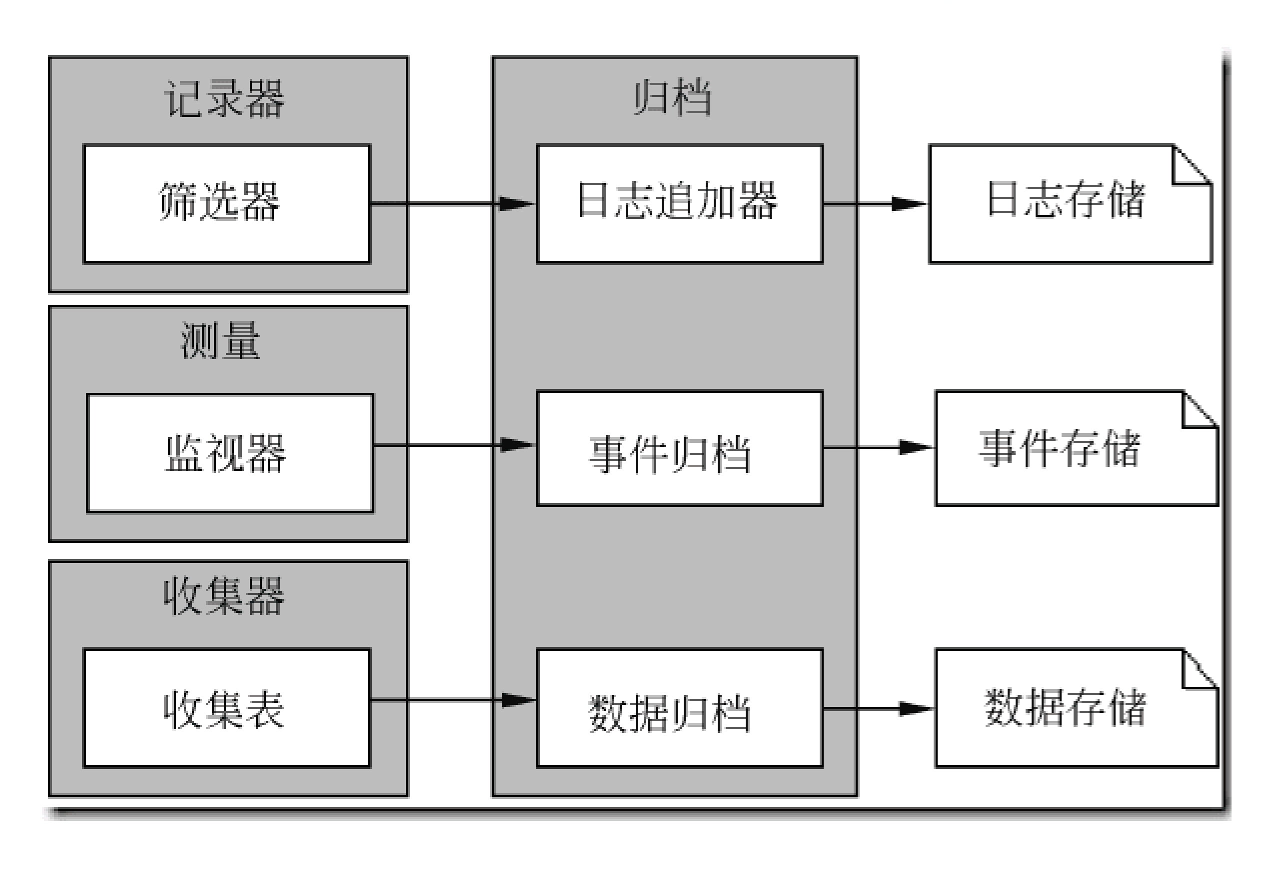


图 33-9

4. 访问器组件（图 33-10）

- 访问器可访问 WLDF 收集到的所有数据（包括日志、事件和规格数据）。访问器会与归档文件进行交互以获取历史数据（包括已记录的事件数据和持久保存的规格）。
- 访问正在运行的服务器中的数据时，会使用基于 JMX 的访问服务。访问器可提供按类型、按组件以及按特性的数据查找。它允许基于时间进行筛选，并且允许按严重程度、来源和内容进行事件筛选。

- 该工具可能会希望访问由当前不处于活动状态的服务器持久保存的数据。在这些情况下还提供了脱机访问器。可使用它将归档的数据导出到一个 XML 文件中以供以后访问。要以此方式使用访问器，必须使用 WebLogic 脚本工具（WebLogic Scripting Tool，WLST）来访问计算机，并且必须具有访问该计算机的物理权限。

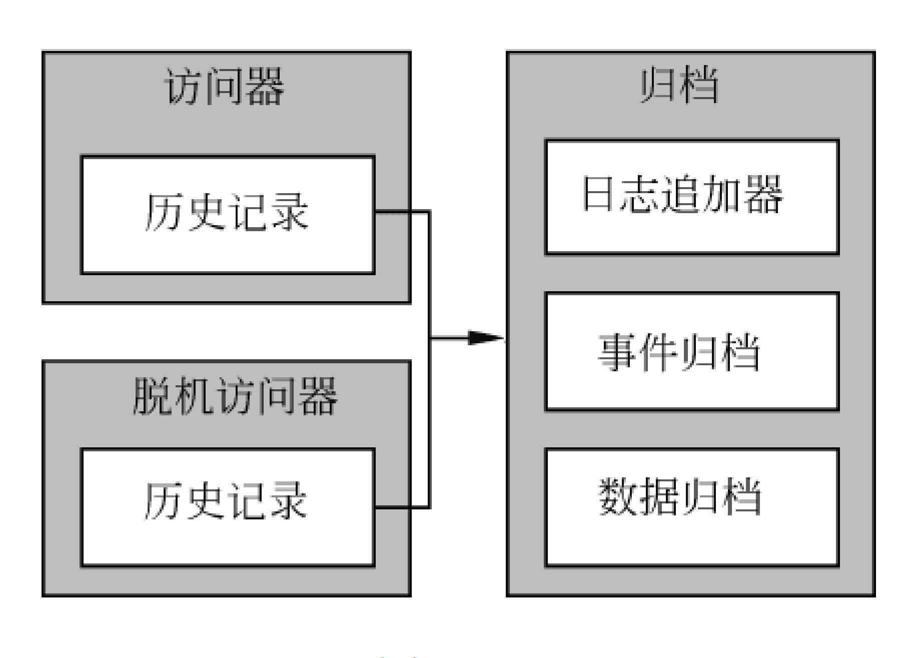


图 33-10

5. 观察器和通知组件（图 30-11）

- 观察器和通知系统可用于创建自动监视器，这些监视器可观察特定诊断状态并基于已配置的规则发送通知。
- 观察器规则可监视测量组件中的日志数据、事件数据，或监视由收集器收集到的数据提供程序中的规格数据。观察器管理器能够管理包含许多观察器规则的观察器。

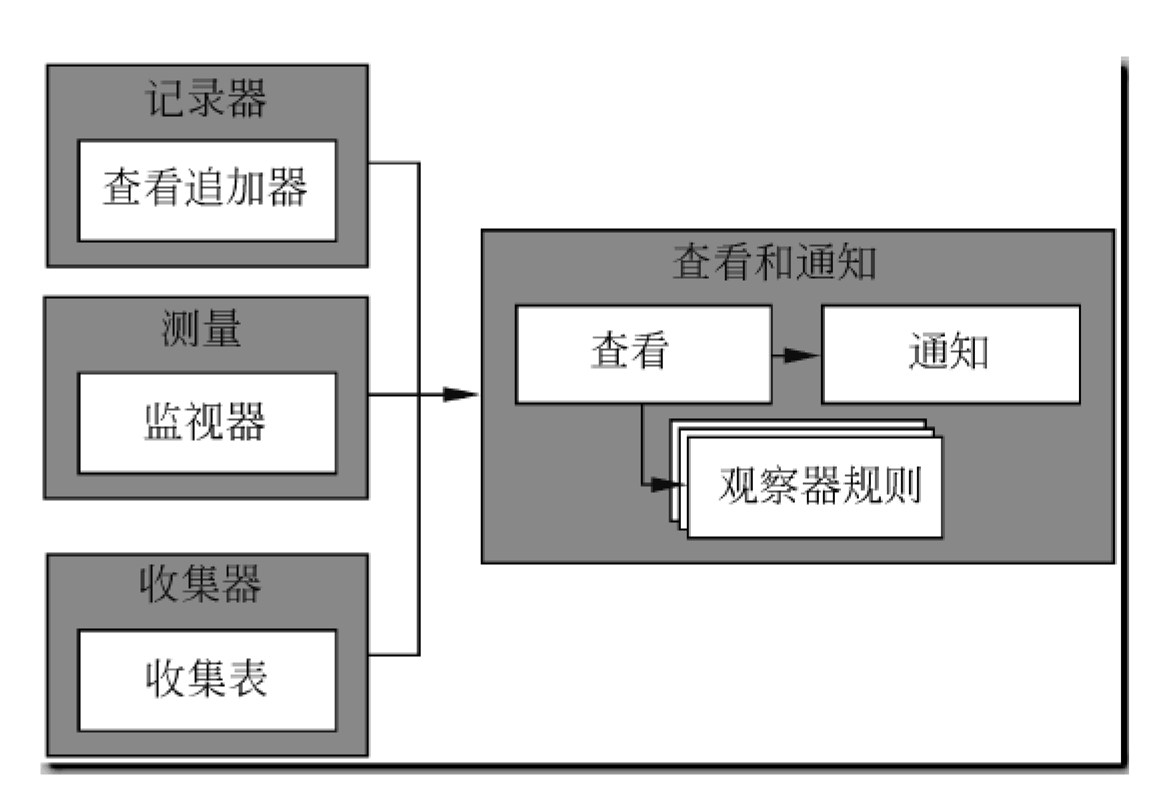


图 33-11

- 可配置一个或多个通知以供观察器使用。默认情况下，每个观察器都会在服务器日志中记录一个事件。另外，还支持 SMTP、SNMP、JMX 和 JMS 通知。
- 数据创建程序可以生成由记录器和收集器使用的诊断数据，如图 33-12 所示。
- 这些组件会与归档文件协同工作来持久保存数据，并且会与观察器和通知子系统协同工作来提供自动监视。访问器会与记录器和收集器进行交互以公开当前的诊断数据，并会与归档文件进行交互提供历史数据。图像捕获工具提供了捕获关键服务器状态诊断快照的方法。

- 所有框架组件都是在服务器级运行的，并且仅知晓服务器作用域。所有组件都完全存在于服务器进程中并参与标准的服务器生命周期。该框架的所有工件都是以每台服务器为基础进行配置和存储的。

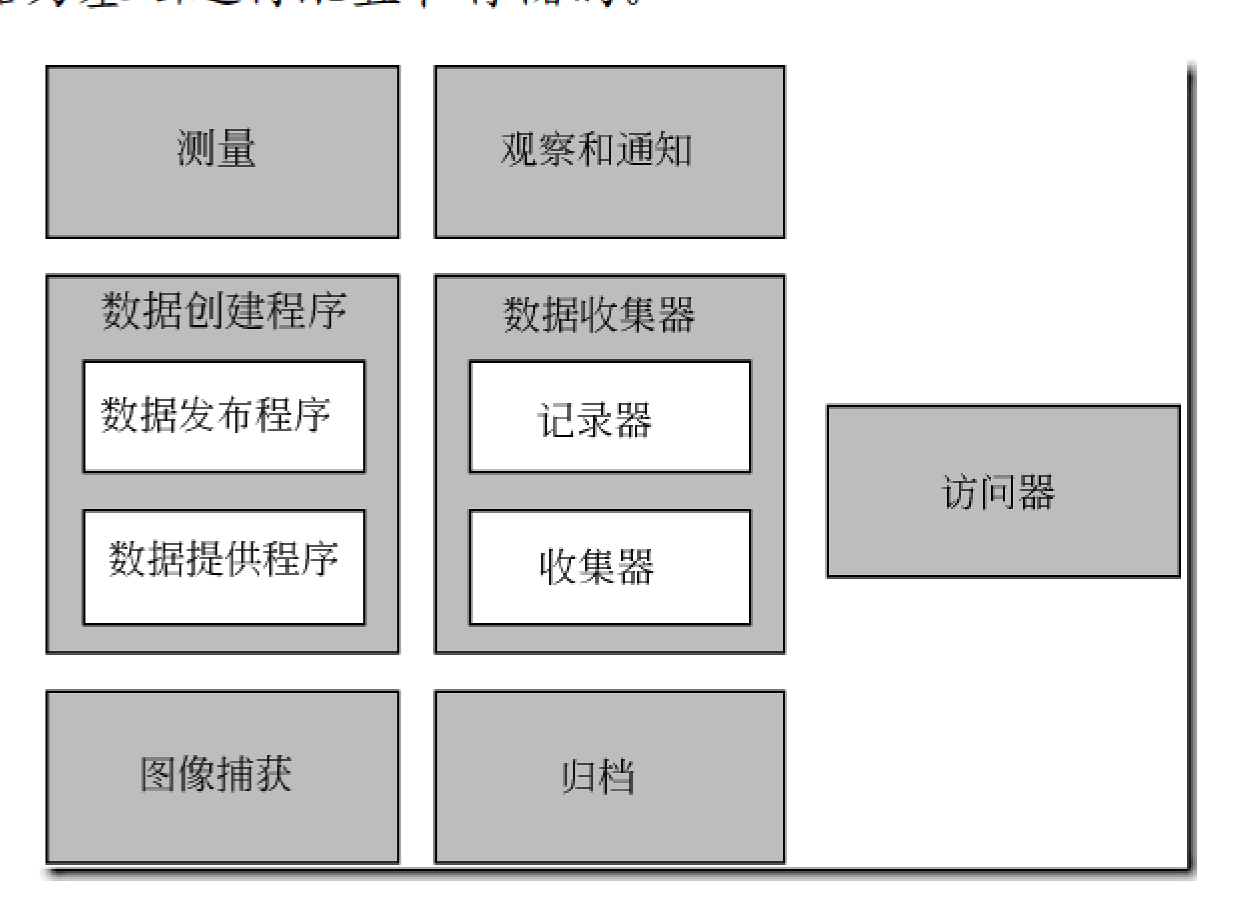


图 33-12

6. 诊断图像捕获

- 诊断图像支持收集了用于诊断问题的关键服务器状态的最常见来源，并将该状态打包到可在支持中使用的单个工件中。诊断图像本质上是来自服务器的诊断快照或转储，类似于 UNIX“核心”转储。
- 图像捕获支持包括按需执行的捕获进程和基于某些基本故障检测的自动化捕获，如图 33-13 所示。

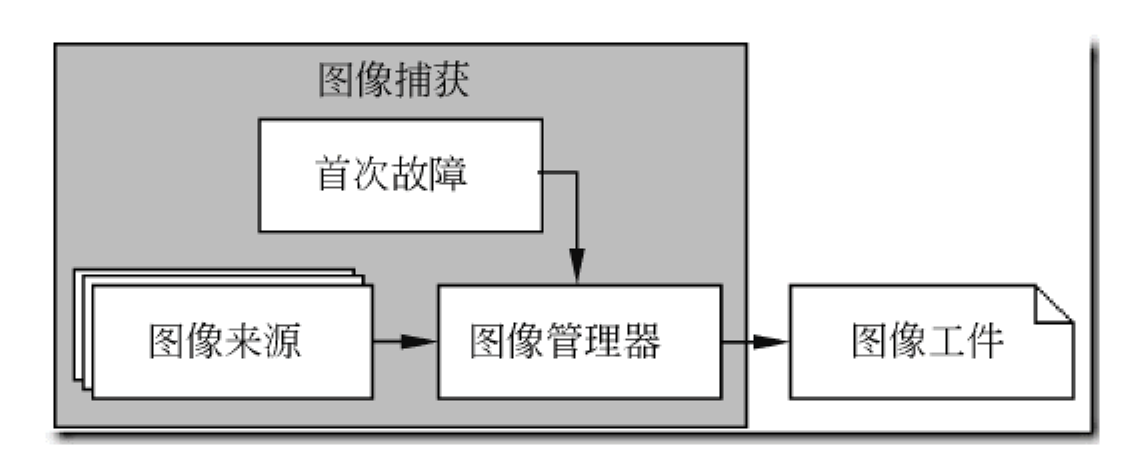


图 33-13

7. 所有部分如何组合在一起（图 33-14）

33.4.3　用 WLDF 运行一个 demo

准备工作如下。

（1）在 Oracle 的官方网站上下载 JRockit Realtime 的最新版本：jrrt-4.0.0-1.6.0-windows-ia32.exe，安装后在其安装目录下可以找到 jrmc.exe，单击就可以运行它的控制台。

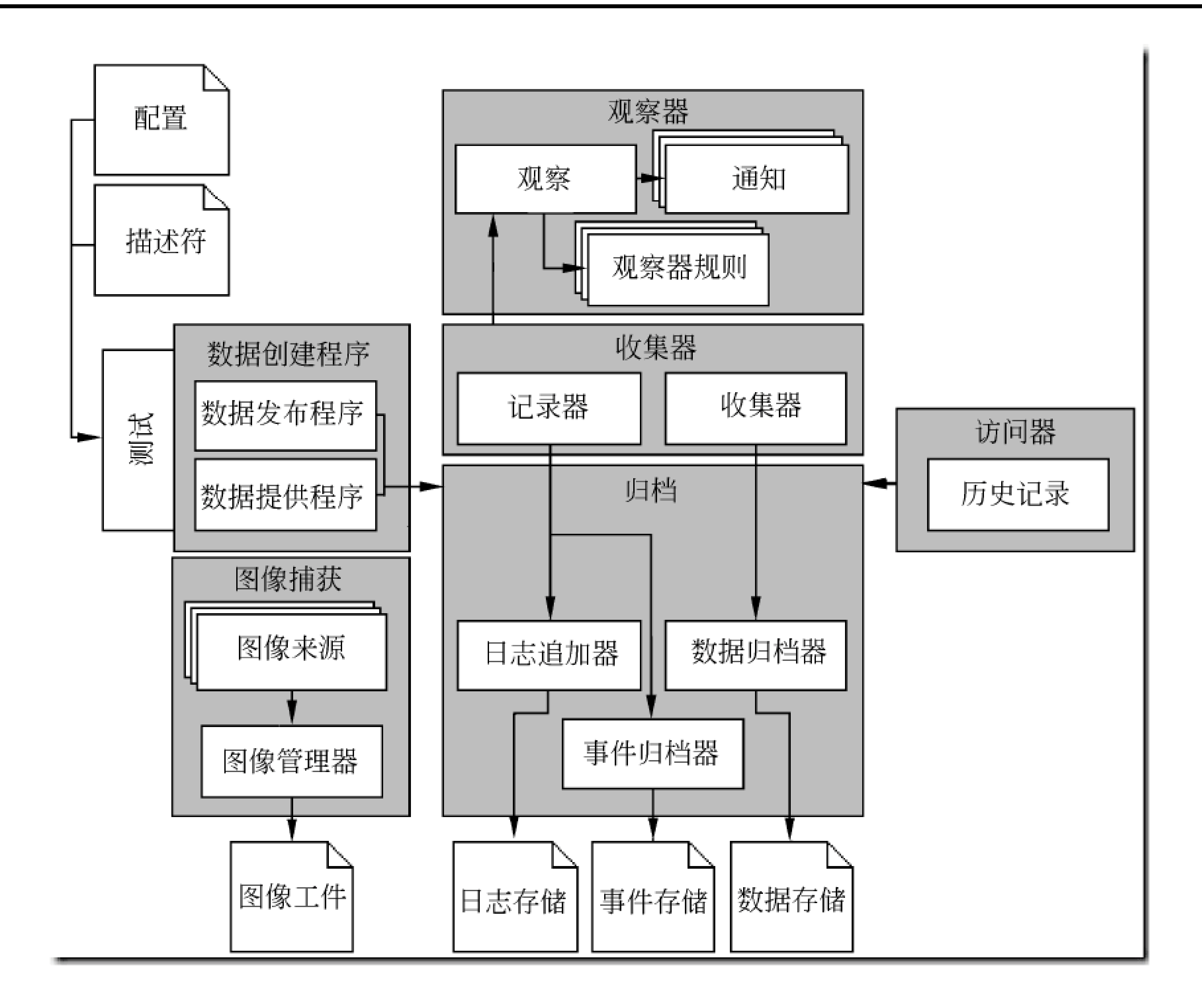


图 33-14

（2）在 WebLogic 中可以单击我们要分析的 Server，如图 33-15 所示。

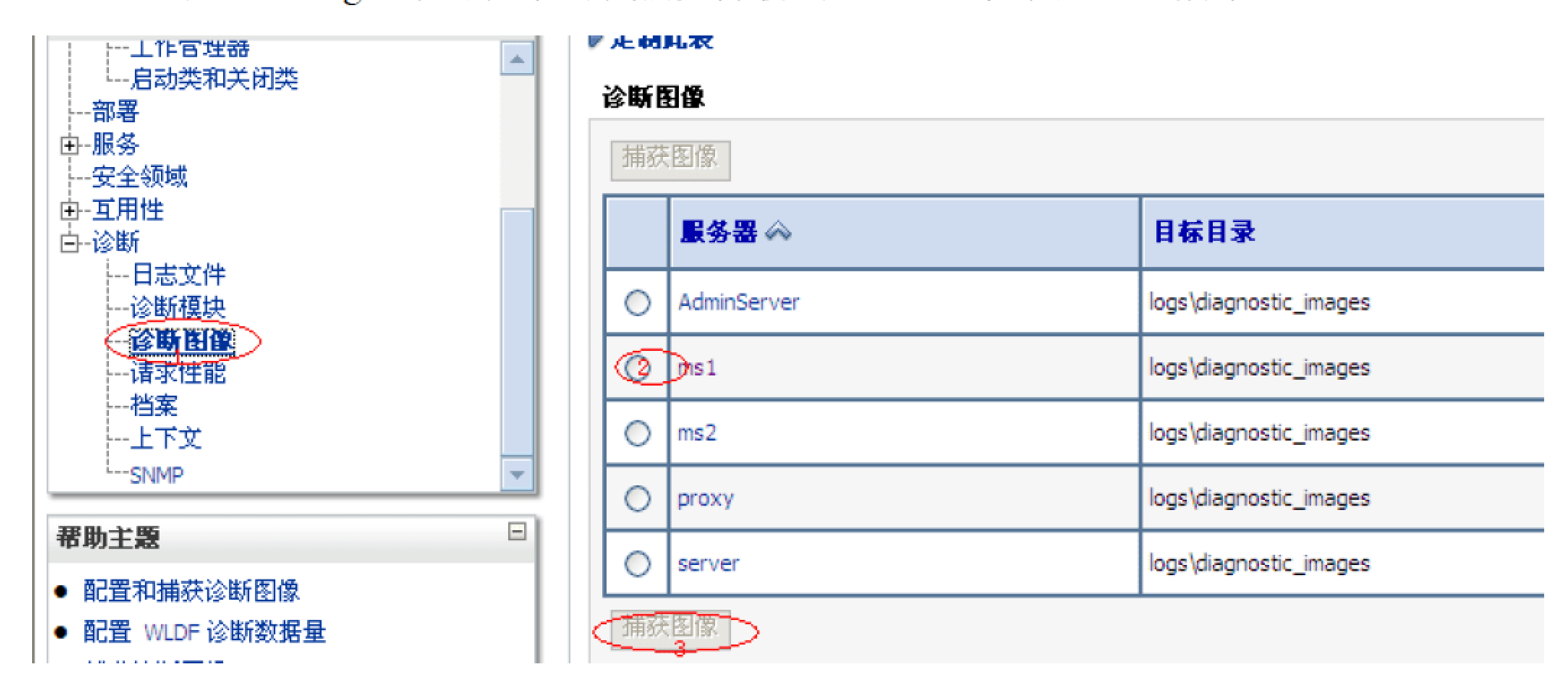


图 33-15

单击“捕获图像”按钮，这样我们就捕获了当前 Server 的一个快照（路径是在 logs\diagnostic\images）。这样就可以把当前的快照导入到 jrmc 中进行分析了，如图 33-16 所示。

在这里我们就可以看到当前系统的内存使用情况，如图 33-17 所示。



图 33-16



图 33-17

CPU 使用情况如图 33-18 所示。

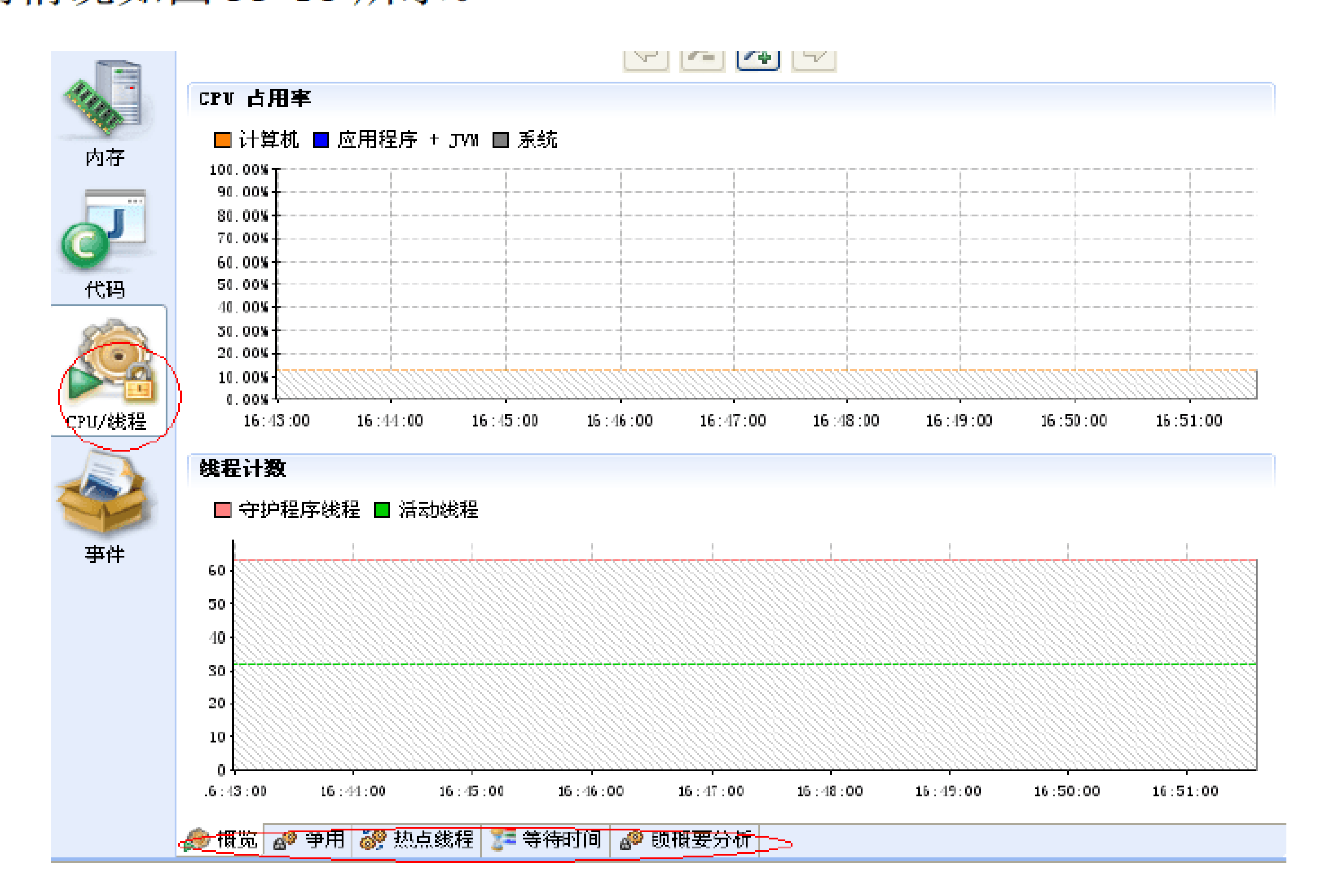


图 33-18

后　　记

作为企业级 IT 运维系列中的一本，《WebLogic 企业级运维实战》如果能抛砖引玉，带给大家一些思路和启迪，则足以欣慰。就像我们回溯历史经历和切身之事时，经常感觉自身的渺小，就如同地球之比于宇宙，小得像太平洋中的一滴水。

而这个世界上，最接近永远不变的，恰恰是“变化”本身。随着产品的更新，新技术的出现，以及厂商之间的商业整合，很多具体的技术细节都会随之发生变化，届时也会跟随有相应的修订；但其中涉及的基本概念和技术思路却是相通的。道为本，术为末，方能根深叶茂。

更多实战经验书籍，请参看叱咤风云同系列的《Tuxedo 企业级运维实战》和《GoldenGate 企业级运维实战》等。